Table of Contents



laravel 源码详解

`laravel` 是一个非常简洁、优雅的 `PHP` 开发框架。`laravel` 中除了提供最为中心的 `Ioc` 容器之外，还提供了强大的 `路由` 、`数据库模型` 等常用功能模块。

对于开发者来说，在使用 `laravel` 框架进行 `web` 开发的同时，一定很好奇 `laravel` 内部各个模块的原理，知其然更知其所以然，有助于提供开发的稳定与效率。

本项目针对 `laravel 5.4` 各个重要模块的源码进行了较为详尽的分析，希望给想要了解 `laravel` 底层原理与源码的同学一些指引。

由于本人能力有限，文章中可能会有一些问题，敬请提出意见与建议，谢谢。

GitBook 地址 ：https://www.gitbook.com/book/leoyang90/laravel-source-analysis

前言

这篇文章是对 `PHP自动加载功能` 的一个总结，内容涉及 `PHP自动加载功能` 、`PHP命名空间` 、`PSR0/PSR4标准` 等内容。

一、PHP 自动加载功能

PHP 自动加载功能的由来

在 PHP 开发过程中，如果希望从外部引入一个 Class ，通常会使用 `include` 和 `require` 方法，去把定义这个 Class 的文件包含进来。这个在小规模开发的时候，没什么大问题。但在大型的开发项目中，使用这种方式会带来一些隐含的问题：如果一个 PHP 文件需要使用很多其它类，那么就需要很多的 `require/include` 语句，这样有可能会 造成遗漏 或者 包含进不必要的类文件。如果大量的文件都需要使用其它的类，那么要保证每个文件都包含正确的类文件肯定是一个噩梦， 况且 require_once 的代价很大。

PHP5 为这个问题提供了一个解决方案，这就是 `类的自动加载(autoload)机制` 。`autoload机制` 可以使得 PHP 程序有可能在使用类时才自动包含类文件，而不是一开始就将所有的类文件 `include` 进来，这种机制也称为 `Lazy loading (延迟加载)` 。

- 总结起来，自动加载功能带来了几处优点：

> 1. 使用类之前无需 `include / require`
> 2. 使用类的时候才会 `include / require` 文件，实现了 `lazy loading` ，避免了 `include / require` 多余文件。
> 3. 无需考虑引入 类的实际磁盘地址 ，实现了逻辑和实体文件的分离。

- 如果想具体详细的了解关于自动加载的功能，可以查看资料：

> PHP 类自动加载机制
>
> PHP autoload机制的实现解析

PHP 自动加载函数 __autoload()

- 通常 PHP5 在使用一个类时，如果发现这个类没有加载，就会自动运行 __autoload() 函数，这个函数是我们在程序中自定义的，在这个函数中我们可以加载需要使用的类。下面是个简单的示例：

```
<?php
function __autoload($classname) {
   require_once ($classname . ".class.php");
}
```

- 在我们这个简单的例子中，我们直接将类名加上扩展名 `.class.php` 构成了类文件名，然后使用 `require_once` 将其加载。

> 从这个例子中，我们可以看出 __autoload 至少要做三件事情：
>
> 1. 根据类名确定类文件名；
> 2. 确定类文件所在的磁盘路径 `(在我们的例子是最简单的情况，类与调用它们的PHP程序文件在同一个文件夹下)`；
> 3. 将类从磁盘文件中加载到系统中。

- 第三步最简单，只需要使用 `include / require` 即可。要实现第一步，第二步的功能，必须在开发时约定类名与磁盘文件的映射方法，只有这样我们才能根据类名找到它对应的磁盘文件。

- 当有大量的类文件要包含的时候，我们只要确定相应的规则，然后在 **`__autoload()`** 函数中，将类名与实际的磁盘文件对应起来，就可以实现 **`lazy loading`** 的效果 。从这里我们也可以看出 `__autoload()` 函数的实现中最重要的是类名与实际的磁盘文件映射规则的实现。

__autoload() 函数存在的问题

- 如果在一个系统的实现中，如果需要使用很多其它的类库，这些类库可能是由不同的开发人员编写的， 其类名与实际的磁盘文件的映射规则不尽相同。这时如果要实现类库文件的自动加载，就必须 在**__autoload()**函数中将所有的映射规则全部实现，这样的话 `__autoload()` 函数有可能会非常复杂，甚至无法实现。最后可能会导致 `__autoload()` 函数十分臃肿，这时即便能够实现，也会给将来的维护和系统效率带来很大的负面影响。

- 那么问题出现在哪里呢？问题出现在 **__autoload()** 是全局函数只能定义一次，不够灵活，所以所有的类名与文件名对应的逻辑规则都要在一个函数里面实现，造成这个函数的臃肿。那么如何来解决这个问题呢？答案就是使用一个 **__autoload**调用堆栈，不同的映射关系写到不同的 `__autoload函数` 中去，然后统一注册统一管理，这个就是PHP5引入的 `SPL Autoload` 。

SPL Autoload

- SPL是Standard PHP Library(标准PHP库)的缩写。它是PHP5引入的一个扩展库，其主要功能包括autoload机制的实现及包括各种Iterator接口或类。SPL Autoload具体有几个函数：

> 1. *spl_autoload_register*：注册*__autoload()*函数
> 2. *spl_autoload_unregister*：注销已注册的函数
> 3. *spl_autoload_functions*：返回所有已注册的函数
> 4. *spl_autoload_call*：尝试所有已注册的函数来加载类
> 5. *spl_autoload* ：*__autoload()*的默认实现
> 6. *spl_autoload_extionsions*： 注册并返回*spl_autoload*函数使用的默认文件扩展名。

todo Continue

todo Continue

todo Continue

todo Continue

todo Continue

todo Continue

这几个函数具体详细用法可见 php中spl_autoload详解

简单来说，spl_autoload 就是 SPL 自己的定义 `__autoload()` 函数，功能很简单，就是去注册的目录(由set_include_path设置)找与$classname同名的.php/.inc文件。当然，你也可以指定特定类型的文件，方法是注册扩展名(spl_autoload_extionsions)。

而 spl*autoload_register()* 就是我们上面所说的_autoload调用堆栈，我们可以向这个函数注册多个我们自己的_autoload()函数，当PHP找不到类名时，PHP就会调用这个堆栈，一个一个去调用自定义的_autoload()函数，实现自动加载功能。如果我们不向这个函数输入任何参数，那么就会注册spl_autoload()函数。

好啦，PHP自动加载的底层就是这些，注册机制已经非常灵活，但是还缺什么呢？我们上面说过，自动加载关键就是类名和文件的映射，这种映射关系不同框架有不同方法，非常灵活，但是过于灵活就会显得杂乱，PHP有专门对这种映射关系的规范，那就是PSR标准中PSR0与PSR4。

不过在谈PSR0与PSR4之前，我们还需要了解PHP的命名空间的问题，因为这两个标准其实针对的都不是类名与目录文件的映射，而是命名空间与文件的映射。为什么会这样呢？在我的理解中，规范的面向对象PHP思想，命名空间在一定程度上算是类名的别名，那么为什么要推出命名空间，命名空间的优点是什么呢

二、Namespace命名空间

要了解命名空间，首先先看看官方文档 中对命名空间的介绍：

> 什么是命名空间？从广义上来说，命名空间是一种封装事物的方法。在很多地方都可以见到这种抽象概念。例如，在操作系统中目录用来将相关文件分组，对于目录中的文件来说，它就扮演了命名空间的角色。具体举个例子，文件 `foo.txt` 可以同时在目录 `/home/greg` 和 `/home/other` 中存在，但在同一个目录中不能存在两个 foo.txt 文件。另外，在目录 `/home/greg` 外访问 `foo.txt` 文件时，我们必须将目录名以及目录分隔符放在文件名之前得到 `/home/greg/foo.txt` 。这个原理应用到程序设计领域就是命名空间的概念。
>
> 在**PHP**中，命名空间用来解决在编写类库或应用程序时创建可重用的代码如类或函数时碰到的两类问题：
>
> 1. 用户编写的代码与PHP内部的类/函数/常量或第三方类/函数/常量之间的名字冲突 *
>
> 2. 为很长的标识符名称(通常是为了缓解第一类问题而定义的)创建一个别名（或简短）的名称，提高源代码的可读性。*
>
> ***PHP*** 命名空间提供了一种将相关的类、函数和常量组合到一起的途径。

简单来说就是PHP是不允许程序中存在两个名字一样一样的类或者函数或者变量名的，那么有人就很疑惑了，那就不起一样名字不就可以了？事实上很多大程序依赖很多第三方库，名字冲突什么的不要太常见，这个就是官网中的第一个问题。那么如何解决这个问题呢？在没有命名空间的时候，可怜的程序员只能给类名起 `a_b_c_d_e_f` 这样的，其中 `a/b/c/d/e/f` 一般有其特定意义，这样一般就不会发生冲突了，但是这样长的类名编写起来累，读起来更是难受。因此PHP5就推出了命名空间，类名是类名，命名空间是命名空间，程序写/看的时候直接用类名，运行起来机器看的是命名空间，这样就解决了问题。

另外，命名空间提供了一种将相关的类、函数和常量组合到一起的途径。这也是面向对象语言命名空间的很大用途，把特定用途所需要的类、变量、函数写到一个命名空间中，进行封装。

解决了类名的问题，我们终于可以回到PSR标准来了，那么PSR0与PSR4是怎么 规范 文件与命名空间的映射关系的呢？答案就是：对 命名空间的命名（额，有点绕）、类文件目录的位置 和 两者映射关系 做出了限制，这个就是标准的核心了。

更完整的描述可见现代 PHP 新特性系列（一） —— 命名空间

三、PSR标准

在说PSR0与PSR4之前先介绍一下PSR标准。PSR标准的发明者和规范者是：PHP-FIG，它的网站是：www.php-fig.org。就是这个联盟组织发明和创造了PSR-[0-4]规范。FIG 是 Framework Interoperability Group（框架可互用性小组）的缩写，由几位开源框架的开发者成立于 2009 年，从那开始也选取了很多其他成员进来，虽然不是“官方”组织，但也代表了社区中不小的一块。组织的目的在于：以最低程度的限制，来统一各个项目的编码规范，避免各家自行发展的风格阻碍了程序设计师开发的困扰，于是大伙发明和总结了PSR，PSR是Proposing a Standards Recommendation（提出标准建议）的缩写，截止到目前为止，总共有5套PSR规范，分别是：

> PSR-0 (Autoloading Standard) 自动加载标准
>
> PSR-1 (Basic Coding Standard)基础编码标准
>
> PSR-2 (Coding Style Guide) 编码风格向导
>
> PSR-3 (Logger Interface) 日志接口
>
> PSR-4 (Improved Autoloading) 自动加载的增强版，可以替换掉*PSR-0*了。

具体详细的规范标准可以查看PHP中PSR-[0-4]规范

PSR0标准

PRS-0规范是他们出的第1套规范，主要是制定了一些自动加载标准（Autoloading Standard）PSR-0强制性要求几点：

> 1. 一个完全合格的namespace和class必须符合这样的结构：`<Vendor Name>[<Namespace>]*<Class Name>`
> 2. 每个namespace必须有一个顶层的namespace（"Vendor Name"提供者名字）
> 3. 每个namespace可以有多个子namespace
> 4. 当从文件系统中加载时，每个namespace的分隔符(/)要转换成 DIRECTORY_SEPARATOR(操作系统路径分隔符)
> 5. 在类名中，每个下划线(_) 符号要转换成 `DIRECTORY_SEPARATOR(操作系统路径分隔符)` 。在 `namespace` 中，下划线 _ 符号是没有（特殊）意义的。
> 6. 当从文件系统中载入时，合格的 `namespace` 和 `class` 一定是以 `.php` 结尾的
> 7. `verdor name` , `namespaces` , `class` 名可以由大小写字母组合而成（大小写敏感的）

具体规则可能有些让人晕，我们从头讲一下。

我们先来看PSR0标准大致内容，第1、2、3、7条对命名空间的名字做出了限制，第4、5条对命名空间和文件目录的映射关系做出了限制，第6条是文件后缀名。

前面我们说过，PSR标准是如何规范命名空间和所在文件目录之间的映射关系？是通过限制命名空间的名字、所在文件目录的位置和两者映射关系。

那么我们可能就要问了，哪里限制了文件所在目录的位置了呢？其实答案就是：

> 限制命名空间名字 `+` 限制命名空间名字与文件目录映射 `=` 限制文件目录

好了，我们先想一想，对于一个具体程序来说，如果它想要支持PSR0标准,它需要做什么调整呢？

> 1. 首先，程序必须定义一个符合PSR0标准第4、5条的映射函数，然后把这个函数注册到spl_register()中；
> 2. 其次，定义一个新的命名空间时，命名空间的名字和所在文件的目录位置必须符合第1、2、3、7条。

一般为了代码维护方便，我们会在一个文件只定义一个命名空间。

好了，我们有了符合PSR0的命名空间的名字，通过符合PSR0标准的映射关系就可以得到符合PSR0标准的文件目录地址，如果我们按照PSR0标准正确存放文件，就可以顺利require该文件了，我们就可以使用该命名空间啦，是不是很神奇呢？

接下来，我们详细地来看看PSR0标准到底规范了什么呢？

我们以laravel中第三方库Symfony其中一个命名空间 `/Symfony/Core/Request` 为例，讲一讲上面PSR0标准。

> 1. 一个完全合格的namespace和class必须符合这样的结构： `<Vendor Name>[<Namespace>]*<Class Name>`

上面所展示的/Symfony就是Vendor Name，也就是第三方库的名字，/Core是Namespace名字，一般是我们命名空间的一些属性信息(例如request是Symfony的核心功能)；最后Request就是我们命名空间的名字，这个标准规范就是让人看到命名空间的来源、功能非常明朗，有利于代码的维护。

> 2 . 每个namespace必须有一个顶层的namespace（"Vendor Name"提供者名字）

也就是说每个命名空间都要有一个类似于/Symfony的顶级命名空间，为什么要有这种规则呢？因为PSR0标准只负责顶级命名空间之后的映射关系，也就是/Symfony/Core/Request这一部分，关于/Symfony应该关联到哪个目录，那就是用户或者框架自己定义的了。所谓的顶层的namespace，就是自定义了映射关系的命名空间，一般就是提供者名字（第三方库的名字）。换句话说顶级命名空间是自动加载的基础。为什么标准要这么设置呢？原因很简单，如果有个命名空间是/Symfony/Core/Transport/Request，还有个命名空间是/Symfony/Core/Transport/Request1,如果没有顶级命名空间，我们就得写两个路径和这两个命名空间相对应，如果再有Request2、Request3呢。有了顶层命名空间/Symfony，那我们就仅仅需要一个目录对应即可，剩下的就利用PSR标准去解析就行了。

> 3.每个namespace可以有多个子namespace

这个很简单，Request可以定义成/Symfony/Core/Request，也可以定义成/Symfony/Core/Transport/Request，/Core这个命名空间下面可以有很多子命名空间，放多少层命名空间都是自己定义。

> 4.当从文件系统中加载时，每个namespace的分隔符(/)要转换成DIRECTORY_SEPARATOR(操作系统路径分隔符)

现在我们终于来到了映射规范了。命名空间的/符号要转为路径分隔符，也就是说要把/Symfony/Core/Request这个命名空间转为\Symfony\Core\Request这样的目录结构。

> 5.在类名中，每个下划线_符号要转换成DIRECTORYSEPARATOR(操作系统路径分隔符)。在namespace中，下划线\符号是没有（特殊）意义的。

这句话的意思就是说，如果我们的命名空间是/Symfony/Core/Request_a，那么我们就应该把它映射到\Symfony\Core\Request\a这样的目录。为什么会有这种规定呢？这是因为PHP5之前并没有命名空间，程序员只能把名字起成Symfony_Core_Request_a这样，PSR0的这条规定就是为了兼容这种情况。

剩下两个很简单就不说了。

有这样的命名空间命名规则和映射标准，我们就可以推理出我们应该把命名空间所在的文件该放在哪里了。依旧以Symfony/Core/Request为例， 它的目录是/path/to/project/vendor/Symfony/Core/Request.php，其中/path/to/project是你项目在磁盘的位置，/path/to/project/vendor是项目用的所有第三方库所在目录。/path/to/project/vendor/Symfony就是与顶级命名空间/Symfony存在对应关系的目录，再往下的文件目录就是按照PSR0标准建立的：

> /Symfony/Core/Request => /Symfony/Core/Request.php

一切很完满了是吗？不，还有一些瑕疵：

> 1. 我们是否应该还兼容没有命名空间的情况呢？
> 2. 按照PSR0标准，命名空间/A/B/C/D/E/F必然对应一个目录结构/A/B/C/D/E/F，这种目录结构层次是不是太深了？

PSR4标准

2013年底，新出了第5个规范——PSR-4。

PSR-4规范了如何指定文件路径从而自动加载类定义，同时规范了自动加载文件的位置。这个乍一看和PSR-0重复了，实际上，在功能上确实有所重复。区别在于PSR-4的规范比较干净，去除了兼容PHP 5.3以前版本的内容，有一点PSR-0升级版的感觉。当然，PSR-4也不是要完全替代PSR-0，而是在必要的时候补充PSR-0——当然，如果你愿意，PSR-4也可以替代PSR-0。PSR-4可以和包括PSR-0在内的其他自动加载机制共同使用。

PSR4标准与PSR0标准的区别：

> 1. 在类名中使用下划线没有任何特殊含义。
> 2. 命名空间与文件目录的映射方法有所调整。

对第二项我们详细解释一下(Composer自动加载的原理)： 假如我们有一个命名空间：Foo/class，Foo是顶级命名空间，其存在着用户定义的与目录的映射关系：

```
"Foo/" => "src/"
```

按照PSR0标准，映射后的文件目录是:src/Foo/class.php，但是按照PSR4标准，映射后的文件目录就会是:src/class.php，为什么要这么更改呢？原因就是怕命名空间太长导致目录层次太深，使得命名空间和文件目录的映射关系更加灵活。

再举一个例子,来源PSR-4——新鲜出炉的PHP规范：

PSR-0风格

```
<?php
-vendor/
| -vendor_name/
| | -package_name/
| | | -src/
| | | | -Vendor_Name/
| | | | | -Package_Name/
| | | | | | -ClassName.php       # Vendor_Name\Package_Name\Clas
sName
| | | -tests/
| | | | -Vendor_Name/
| | | | | -Package_Name/
| | | | | | -ClassNameTest.php   # Vendor_Name\Package_Name\Clas
sName
```

PSR-4风格

```
-vendor/
| -vendor_name/
| | -package_name/
| | | -src/
| | | | -ClassName.php       # Vendor_Name\Package_Name\ClassName

| | | -tests/
| | | | -ClassNameTest.php   # Vendor_Name\Package_Name\ClassNam
eTest
```

对比以上两种结构，明显可以看出PSR-4带来更简洁的文件结构。

Written with StackEdit.

前言

上一篇文章中，我们讨论了 `PHP自动加载功能` 、 `PHP命名空间` 、 `PSR0/PSR4标准` ，有了这些知识，其实我们就可以按照 PSR4标准 写出可以自动加载的程序了。然而我们为什么要自己写呢？尤其是有 `Composer` 这神一样的包管理器的情况下？

Composer自动加载概论

简介

`Composer` 是 PHP 的一个依赖管理工具。它允许你申明项目所依赖的代码库，它会在你的项目中为你安装他们。详细内容可以查看Composer 中文网。

Composer Composer 将这样为你解决问题：

> - 你有一个项目依赖于若干个库。
> - 其中一些库依赖于其他库。
> - 你声明你所依赖的东西。
> - Composer 会找出哪个版本的包需要安装，并安装它们（将它们下载到你的项目中）。

例如，你正在创建一个项目，你需要一个库来做日志记录。你决定使用 `monolog` 。为了将它添加到你的项目中，你所需要做的就是创建一个 `composer.json` 文件，其中描述了项目的依赖关系。

```
{
  "require": {
    "monolog/monolog": "1.2.*"
  }
}
```

然后我们只要在项目里面直接 `use Monolog\Logger` 即可，神奇吧！

简单的说，Composer 帮助我们下载好了符合 `PSR0/PSR4标准` 的第三方库，并把文件放在相应位置；帮我们写了 `__autoload()` 函数，注册到了 `spl_register()` 函数，当我们想用第三方库的时候直接使用命名空间即可。

那么当我们想要写自己的命名空间的时候，该怎么办呢？很简单，我们只要按照 PSR4标准 命名我们的命名空间，放置我们的文件，然后在 composer 里面写好顶级域名与具体目录的映射，就可以享用 composer 的便利了。

当然如果有一个非常棒的框架，我们会惊喜地发现，在 composer 里面写顶级域名映射这事我们也不用做了，框架已经帮我们写好了顶级域名映射了，我们只需要在框架里面新建文件，在新建的文件中写好命名空间，就可以在任何地方 use 我们的命名空间了。

下面我们就以 Laravel 框架为例，讲一讲 composer 是如何实现 `PSR0/PSR4标准` 的自动加载功能。

Composer自动加载文件

首先，我们先大致了解一下Composer自动加载所用到的源文件。

> 1. autoload_real.php: 自动加载功能的引导类。
>   - 任务是composer加载类的初始化`（顶级命名空间与文件路径映射初始化）`和注册(spl_autoload_register())。
> 2. ClassLoader.php: composer加载类。
>   - composer自动加载功能的核心类。
> 3. autoload_static.php: 顶级命名空间初始化类，
>   - 用于给核心类初始化顶级命名空间。
> 4. autoload_classmap.php: 自动加载的最简单形式，
>   - 有完整的命名空间和文件目录的映射；
> 5. autoload_files.php: 用于加载全局函数的文件，
>   - 存放各个全局函数所在的文件路径名；
> 6. autoload_namespaces.php: 符合PSR0标准的自动加载文件，
>   - 存放着顶级命名空间与文件的映射；
> 7. autoload_psr4.php: 符合PSR4标准的自动加载文件，
>   - 存放着顶级命名空间与文件的映射；

laravel框架下Composer的自动加载源码分析

——启动

laravel框架的初始化是需要composer自动加载协助的，所以laravel的入口文件index.php第一句就是利用composer来实现自动加载功能。

```
<?php
   require __DIR__.'/../bootstrap/autoload.php';
```

咱们接着去看 bootstrap 目录下的 `autoload.php` ：

```
<?php
  define('LARAVEL_START', microtime(true));

  require __DIR__ . '/../vendor/autoload.php';
```

再去 vendor 目录下的 `autoload.php` ：

```
<?php
  require_once __DIR__ . '/composer' . '/autoload_real.php';

  return ComposerAutoloaderInit832ea71bfb9a4128da8660baedaac82e::getLoader();
```

为什么框架要在 `bootstrap/autoload.php` 转一下？个人理解，laravel 这样设计有利于支持或扩展任意有自动加载的第三方库。

好了，我们终于要看到了Composer真正要显威的地方了。 `autoload_real.php` 里面就是一个自动加载功能的引导类，这个类不负责具体功能逻辑，只做了两件事：初始化自动加载类、注册自动加载类。

到autoload_real这个文件里面去看，发现这个引导类的名字叫ComposerAutoloaderInit832ea71bfb9a4128da8660baedaac82e，为什么要叫这么古怪的名字呢？因为这是防止用户自定义类名跟这个类重复冲突了，所以在类名上加了一个hash值。其实还有一个做法我们更加熟悉，那就是不直接定义类名，而是

定义一个命名空间。这里为什么不定义一个命名空间呢？个人理解：命名空间一般都是为了复用，而这个类只需要运行一次即可，以后也不会用得到，用hash值更加合适。

laravel框架下Composer的自动加载源码分析——autoload_real引导类

在 vendor 目录下的 `autoload.php` 文件中我们可以看出，程序主要调用了引导类的静态方法 `getLoader()` ，我们接着看看这个函数。

```
<?php
    public static function getLoader()
    {
      /***************************经典单例模式********************/

      if (null !== self::$loader) {
          return self::$loader;
      }

      /***********************获得自动加载核心类对象****************
****/
      spl_autoload_register(
        array('ComposerAutoloaderInit832ea71bfb9a4128da8660baeda
ac82e', 'loadClassLoader'), true, true
      );

      self::$loader = $loader = new \Composer\Autoload\ClassLoad
er();

      spl_autoload_unregister(
        array('ComposerAutoloaderInit832ea71bfb9a4128da8660baeda
ac82e', 'loadClassLoader')
      );

      /***********************初始化自动加载核心类对象**************
******/
```

```
        $useStaticLoader = PHP_VERSION_ID >= 50600 && !defined('HH
VM_VERSION');

        if ($useStaticLoader) {
            require_once __DIR__ . '/autoload_static.php';

            call_user_func(
            \Composer\Autoload\ComposerStaticInit832ea71bfb9a4128d
a8660baedaac82e::getInitializer($loader)
            );

        } else {
            $map = require __DIR__ . '/autoload_namespaces.php';
            foreach ($map as $namespace => $path) {
                $loader->set($namespace, $path);
            }

            $map = require __DIR__ . '/autoload_psr4.php';
            foreach ($map as $namespace => $path) {
                $loader->setPsr4($namespace, $path);
            }

            $classMap = require __DIR__ . '/autoload_classmap.php'
;
            if ($classMap) {
                $loader->addClassMap($classMap);
            }
        }

        /***********************注册自动加载核心类对象****************
****/
        $loader->register(true);

        /***********************自动加载全局函数********************/
        if ($useStaticLoader) {
            $includeFiles = Composer\Autoload\ComposerStaticInit83
2ea71bfb9a4128da8660baedaac82e::$files;
        } else {
            $includeFiles = require __DIR__ . '/autoload_files.php'
;
```

```
        }

        foreach ($includeFiles as $fileIdentifier => $file) {
            composerRequire832ea71bfb9a4128da8660baedaac82e($fileI
dentifier, $file);
        }

        return $loader;
    }
```

从上面可以看出，我把自动加载引导类分为5个部分。

第一部分——单例

第一部分很简单，就是个最经典的单例模式，自动加载类只能有一个。

```
<?php
  if (null !== self::$loader) {
      return self::$loader;
  }
```

第二部分——构造ClassLoader核心类

第二部分 new 一个自动加载的核心类对象。

```
<?php
  /***********************获得自动加载核心类对象********************/

  spl_autoload_register(
    array('ComposerAutoloaderInit832ea71bfb9a4128da8660baedaac82
e', 'loadClassLoader'), true, true
  );

  self::$loader = $loader = new \Composer\Autoload\ClassLoader()
;

  spl_autoload_unregister(
    array('ComposerAutoloaderInit832ea71bfb9a4128da8660baedaac82
e', 'loadClassLoader')
  );
```

`loadClassLoader()` 函数：

```
<?php
public static function loadClassLoader($class)
{
    if ('Composer\Autoload\ClassLoader' === $class) {
        require __DIR__ . '/ClassLoader.php';
    }
}
```

从程序里面我们可以看出，composer 先向 PHP 自动加载机制注册了一个函数，这个函数 require 了 ClassLoader 文件。成功 new 出该文件中核心类 ClassLoader() 后，又销毁了该函数。

为什么不直接require，而要这么麻烦？原因就是怕有的用户也定义了个 `\Composer\Autoload\ClassLoader` 命名空间，导致自动加载错误文件。那为什么不跟引导类一样用个 hash 呢？因为这个类是可以复用的，框架允许用户使用这个类。

第三部分 —— 初始化核心类对象

```
<?php
  /***********************初始化自动加载核心类对象******************
**/
  $useStaticLoader = PHP_VERSION_ID >= 50600 && !defined('HHVM_V
ERSION');
  if ($useStaticLoader) {
     require_once __DIR__ . '/autoload_static.php';

     call_user_func(
       \Composer\Autoload\ComposerStaticInit832ea71bfb9a4128da86
60baedaac82e::getInitializer($loader)
     );
  } else {
      $map = require __DIR__ . '/autoload_namespaces.php';
      foreach ($map as $namespace => $path) {
         $loader->set($namespace, $path);
      }

      $map = require __DIR__ . '/autoload_psr4.php';
      foreach ($map as $namespace => $path) {
         $loader->setPsr4($namespace, $path);
      }

      $classMap = require __DIR__ . '/autoload_classmap.php';
      if ($classMap) {
          $loader->addClassMap($classMap);
      }
    }
```

这一部分就是对自动加载类的初始化，主要是给自动加载核心类初始化顶级命名空间映射。

初始化的方法有两种：

```
  1. 使用autoload_static进行静态初始化；
  2. 调用核心类接口初始化。
```

autoload_static静态初始化

静态初始化只支持 PHP5.6 以上版本并且不支持 HHVM 虚拟机。我们深入 `autoload_static.php` 这个文件发现这个文件定义了一个用于静态初始化的类，名字叫 `ComposerStaticInit832ea71bfb9a4128da8660baedaac82e`，仍然为了避免冲突加了 hash 值。这个类很简单：

```
<?php
  class ComposerStaticInit832ea71bfb9a4128da8660baedaac82e{
     public static $files = array(...);
     public static $prefixLengthsPsr4 = array(...);
     public static $prefixDirsPsr4 = array(...);
     public static $prefixesPsr0 = array(...);
     public static $classMap = array (...);

    public static function getInitializer(ClassLoader $loader)
    {
      return \Closure::bind(function () use ($loader) {
          $loader->prefixLengthsPsr4
                          = ComposerStaticInit832ea71bfb9a4128da
8660baedaac82e::$prefixLengthsPsr4;

          $loader->prefixDirsPsr4
                          = ComposerStaticInit832ea71bfb9a4128da
8660baedaac82e::$prefixDirsPsr4;

          $loader->prefixesPsr0
                          = ComposerStaticInit832ea71bfb9a4128da
8660baedaac82e::$prefixesPsr0;

          $loader->classMap
                          = ComposerStaticInit832ea71bfb9a4128da
8660baedaac82e::$classMap;

      }, null, ClassLoader::class);
  }
```

这个静态初始化类的核心就是 `getInitializer()` 函数，它将自己类中的顶级命名空间映射给了 ClassLoader类。值得注意的是这个函数返回的是一个匿名函数，为什么呢？原因就是 `ClassLoader类` 中的 `prefixLengthsPsr4`

、 `prefixDirsPsr4` 等等方法都是private的。。。普通的函数没办法给类的 `private` 成员变量赋值。利用匿名函数的绑定功能就可以将把匿名函数转为ClassLoader类 的成员函数。

关于匿名函数的绑定功能。

接下来就是顶级命名空间初始化的关键了。

最简单的 classMap:

```
<?php
  public static $classMap = array (
      'App\\Console\\Kernel'
              => __DIR__ . '/../..' . '/app/Console/Kernel.php',

      'App\\Exceptions\\Handler'
              => __DIR__ . '/../..' . '/app/Exceptions/Handler.p
hp',

      'App\\Http\\Controllers\\Auth\\ForgotPasswordController'
              => __DIR__ . '/../..' . '/app/Http/Controllers/Aut
h/ForgotPasswordController.php',

      'App\\Http\\Controllers\\Auth\\LoginController'
              => __DIR__ . '/../..' . '/app/Http/Controllers/Aut
h/LoginController.php',

      'App\\Http\\Controllers\\Auth\\RegisterController'
              => __DIR__ . '/../..' . '/app/Http/Controllers/Aut
h/RegisterController.php',
  ...)
```

简单吧，直接命名空间全名与目录的映射，没有顶级命名空间。。。简单粗暴，也导致这个数组相当的大。

PSR0 顶级命名空间映射：

```
<?php
  public static $prefixesPsr0 = array (
    'P' => array (
        'Prophecy\\' => array (
            0 => __DIR__ . '/..' . '/phpspec/prophecy/src',
        ),

        'Parsedown' => array (
            0 => __DIR__ . '/..' . '/erusev/parsedown',
        ),
    ),
    'M' => array (
        'Mockery' => array (
            0 => __DIR__ . '/..' . '/mockery/mockery/library',
        ),
    ),
    'J' => array (
        'JakubOnderka\\PhpConsoleHighlighter' => array (
            0 => __DIR__ . '/..' . '/jakub-onderka/php-console-h
ighlighter/src',
        ),
        'JakubOnderka\\PhpConsoleColor' => array (
            0 => __DIR__ . '/..' . '/jakub-onderka/php-console-c
olor/src',
        ),
    ),
    'D' => array (
        'Doctrine\\Common\\Inflector\\' => array (
            0 => __DIR__ . '/..' . '/doctrine/inflector/lib',
        ),
    ),
  );
```

为了快速找到顶级命名空间，我们这里使用命名空间第一个字母作为前缀索引。这个映射的用法比较明显，假如我们有Parsedown/example这样的命名空间，首先通过首字母P，找到

```php
<?php
  'P' => array (
      'Prophecy\\' => array (
        0 => __DIR__ . '/..' . '/phpspec/prophecy/src',
      ),

      'Parsedown' => array (
        0 => __DIR__ . '/..' . '/erusev/parsedown',
      ),
  ),
```

这个数组，然后我们就会遍历这个数组来和 `Parsedown/example` 比较，发现第一个 Prophecy 不符合，第二个 Parsedown 符合，然后得到了映射目录：(映射目录可能不止一个)

```php
<?php
    array (0 => __DIR__ . '/..' . '/erusev/parsedown',)
```

我们会接着遍历这个数组，尝试 `__DIR__ . '/..' . '/erusev/parsedown/Parsedown/example.php'` 是否存在，如果不存在接着遍历数组(这个例子数组只有一个元素)，如果数组遍历完都没有，就会加载失败。

PSR4标准顶级命名空间映射数组：

```
<?php
  public static $prefixLengthsPsr4 = array(
      'p' => array (
        'phpDocumentor\\Reflection\\' => 25,
    ),
      'S' => array (
        'Symfony\\Polyfill\\Mbstring\\' => 26,
        'Symfony\\Component\\Yaml\\' => 23,
        'Symfony\\Component\\VarDumper\\' => 28,
        ...
    ),
  ...);

  public static $prefixDirsPsr4 = array (
      'phpDocumentor\\Reflection\\' => array (
        0 => __DIR__ . '/..' . '/phpdocumentor/reflection-common
/src',
        1 => __DIR__ . '/..' . '/phpdocumentor/type-resolver/src'
,
        2 => __DIR__ . '/..' . '/phpdocumentor/reflection-docblo
ck/src',
    ),
       'Symfony\\Polyfill\\Mbstring\\' => array (
        0 => __DIR__ . '/..' . '/symfony/polyfill-mbstring',
    ),
      'Symfony\\Component\\Yaml\\' => array (
        0 => __DIR__ . '/..' . '/symfony/yaml',
    ),
  ...)
```

PSR4标准顶级命名空间映射用了两个数组，第一个和 PSR0 一样用命名空间第一个字母作为前缀索引，然后是 顶级命名空间，但是最终并不是文件路径，而是 顶级命名空间 的长度。为什么呢？因为前一篇文章我们说过，PSR4标准 的文件目录更加灵活，更加简洁。

PSR0 中顶级命名空间目录直接加到命名空间前面就可以得到路径

```
Parsedown/example => __DIR__ . '/..' . '/erusev/parsedown/Parsed
own/example.php
                                                           ↑↑↑↑↑
↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
```

而 PSR4标准 却是用顶级命名空间目录替换顶级命名空间，所以获得顶级命名空间的长度很重要。

```
Parsedown/example => __DIR__ . '/..' . '/erusev/parsedown/exampl
e.php
                                                ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
↑↑↑↑↑
```

具体的用法：假如我们找 `Symfony\\Polyfill\\Mbstring\\example` 这个命名空间，和 PSR0 一样通过前缀索引和字符串匹配我们得到了

```
<?php
    'Symfony\\Polyfill\\Mbstring\\' => 26,
```

这条记录，键是顶级命名空间，值是命名空间的长度。拿到顶级命名空间后去 `$prefixDirsPsr4数组` 获取它的映射目录数组：(注意映射目录可能不止一条)

```
<?php
  'Symfony\\Polyfill\\Mbstring\\' => array (
            0 => __DIR__ . '/..' . '/symfony/polyfill-mbstring'
,
          )
```

然后我们就可以将命名空间 `Symfony\\Polyfill\\Mbstring\\example` 前26个字符替换成目录 `__DIR__ . '/..' . '/symfony/polyfill-mbstring` ，我们就得到了 `__DIR__ . '/..' . '/symfony/polyfill-mbstring/example.php` ，先验证磁盘上这个文件是否存在，如果不存在接着遍历。如果遍历后没有找到，则加载失败。

自动加载核心类**ClassLoader**的静态初始化完成！！！

ClassLoader接口初始化

如果PHP版本低于5.6或者使用 HHVM 虚拟机环境，那么就要使用核心类的接口进行初始化。

```
<?php
    //PSR0标准
    $map = require __DIR__ . '/autoload_namespaces.php';
    foreach ($map as $namespace => $path) {
       $loader->set($namespace, $path);
    }

    //PSR4标准
    $map = require __DIR__ . '/autoload_psr4.php';
    foreach ($map as $namespace => $path) {
       $loader->setPsr4($namespace, $path);
    }

    $classMap = require __DIR__ . '/autoload_classmap.php';
    if ($classMap) {
       $loader->addClassMap($classMap);
    }
```

PSR0标准

autoload_namespaces：

```
<?php
    return array(
        'Prophecy\\'
            => array($vendorDir . '/phpspec/prophecy/src'),

        'Parsedown'
            => array($vendorDir . '/erusev/parsedown'),

        'Mockery'
            => array($vendorDir . '/mockery/mockery/library'),

        'JakubOnderka\\PhpConsoleHighlighter'
            => array($vendorDir . '/jakub-onderka/php-console-hi
ghlighter/src'),

        'JakubOnderka\\PhpConsoleColor'
            => array($vendorDir . '/jakub-onderka/php-console-co
lor/src'),

        'Doctrine\\Common\\Inflector\\'
            => array($vendorDir . '/doctrine/inflector/lib'),
    );
```

PSR0标准的初始化接口：

```
<?php
    public function set($prefix, $paths)
    {
        if (!$prefix) {
            $this->fallbackDirsPsr0 = (array) $paths;
        } else {
            $this->prefixesPsr0[$prefix[0]][$prefix] = (array) $
paths;
        }
    }
```

很简单，PSR0标准取出命名空间的第一个字母作为索引，一个索引对应多个顶级命名空间，一个顶级命名空间对应多个目录路径，具体形式可以查看上面我们讲的 autoload_static 的 $prefixesPsr0 。如果没有顶级命名空间，就只存储一个路径名，以便在后面尝试加载。

PSR4标准

autoload_psr4

```
<?php
    return array(
    'XdgBaseDir\\'
        => array($vendorDir . '/dnoegel/php-xdg-base-dir/src'),

    'Webmozart\\Assert\\'
        => array($vendorDir . '/webmozart/assert/src'),

    'TijsVerkoyen\\CssToInlineStyles\\'
        => array($vendorDir . '/tijsverkoyen/css-to-inline-style
s/src'),

    'Tests\\'
        => array($baseDir . '/tests'),

    'Symfony\\Polyfill\\Mbstring\\'
        => array($vendorDir . '/symfony/polyfill-mbstring'),
    ...
    )
```

PSR4标准的初始化接口:

```
<?php
    public function setPsr4($prefix, $paths)
    {
        if (!$prefix) {
            $this->fallbackDirsPsr4 = (array) $paths;
        } else {
            $length = strlen($prefix);
            if ('\\' !== $prefix[$length - 1]) {
                throw new \InvalidArgumentException(
                  "A non-empty PSR-4 prefix must end with a name
space separator."
                );
            }
            $this->prefixLengthsPsr4[$prefix[0]][$prefix] = $len
gth;
            $this->prefixDirsPsr4[$prefix] = (array) $paths;
        }
    }
```

PSR4初始化接口也很简单。如果没有顶级命名空间，就直接保存目录。如果有命名空间的话，要保证顶级命名空间最后是 `\` ，然后分别保存

```
( 前缀 -> 顶级命名空间，顶级命名空间 -> 顶级命名空间长度 )
( 顶级命名空间 -> 目录 )
```

这两个映射数组。具体形式可以查看上面我们讲的 `autoload_static` 的 prefixLengthsPsr4 、 $prefixDirsPsr4 。

傻瓜式命名空间映射

autoload_classmap：

```
<?php
public static $classMap = array (
    'App\\Console\\Kernel'
        => __DIR__ . '/../..' . '/app/Console/Kernel.php',

    'App\\Exceptions\\Handler'
        => __DIR__ . '/../..' . '/app/Exceptions/Handler.php',
    ...
)
```

addClassMap:

```
<?php
    public function addClassMap(array $classMap)
    {
        if ($this->classMap) {
            $this->classMap = array_merge($this->classMap, $clas
sMap);
        } else {
            $this->classMap = $classMap;
        }
    }
```

这个最简单，就是整个命名空间与目录之间的映射。

结语

其实我很想接着写下下去，但是这样会造成篇幅过长，所以我就把自动加载的注册和运行放到下一篇文章了。我们回顾一下，这篇文章主要讲了：

```
1.框架如何启动composer自动加载;
2.composer自动加载分为5部分；
```

其实说是5部分，真正重要的就两部分——初始化与注册。初始化负责顶层命名空间的目录映射，注册负责实现顶层以下的命名空间映射规则。

Written with StackEdit.

前言

上一篇文章我们讲到了Composer自动加载功能的启动与初始化，经过启动与初始化，自动加载核心类对象已经获得了顶级命名空间与相应目录的映射，换句话说，如果有命名空间'App\Console\Kernel，我们已经知道了App\对应的目录，接下来我们就要解决下面的就是\Console\Kernel这一段。

Composer自动加载源码分析——注册

我们先回顾一下自动加载引导类：

```
```php
public static function getLoader()
{
    /***************************经典单例模式********************/
    if (null !== self::$loader) {
        return self::$loader;
    }

    /***********************获得自动加载核心类对象******************
**/
    spl_autoload_register(array('ComposerAutoloaderInit
    832ea71bfb9a4128da8660baedaac82e', 'loadClassLoader'), true,
 true);

    self::$loader = $loader = new \Composer\Autoload\ClassLoader
();

    spl_autoload_unregister(array('ComposerAutoloaderInit
    832ea71bfb9a4128da8660baedaac82e', 'loadClassLoader'));

    /***********************初始化自动加载核心类对象****************
****/
```
```

```
    $useStaticLoader = PHP_VERSION_ID >= 50600 &&
    !defined('HHVM_VERSION');

    if ($useStaticLoader) {
        require_once __DIR__ . '/autoload_static.php';

        call_user_func(\Composer\Autoload\ComposerStaticInit
        832ea71bfb9a4128da8660baedaac82e::getInitializer($loader
));

    } else {
        $map = require __DIR__ . '/autoload_namespaces.php';
        foreach ($map as $namespace => $path) {
            $loader->set($namespace, $path);
        }

        $map = require __DIR__ . '/autoload_psr4.php';
        foreach ($map as $namespace => $path) {
            $loader->setPsr4($namespace, $path);
        }

        $classMap = require __DIR__ . '/autoload_classmap.php';
        if ($classMap) {
            $loader->addClassMap($classMap);
        }
    }

    /***********************注册自动加载核心类对象*****************
**/
    $loader->register(true);

    /***********************自动加载全局函数********************/
    if ($useStaticLoader) {
        $includeFiles = Composer\Autoload\ComposerStaticInit
        832ea71bfb9a4128da8660baedaac82e::$files;
    } else {
        $includeFiles = require __DIR__ . '/autoload_files.php';
    }

    foreach ($includeFiles as $fileIdentifier => $file) {
```

```
        composerRequire
        832ea71bfb9a4128da8660baedaac82e($fileIdentifier, $file)
;
    }

    return $loader;
}
  ```
```

现在我们开始引导类的第四部分：注册自动加载核心类对象。我们来看看核心类的register()函数：

```
```php
public function register($prepend = false)
{
    spl_autoload_register(array($this, 'loadClass'), true, $prep
end);
}
```
```

简单到爆炸啊！一行代码实现自动加载有木有！其实奥秘都在自动加载核心类ClassLoader的loadClass()函数上，这个函数负责按照PSR标准将顶层命名空间以下的内容转为对应的目录，也就是上面所说的将'App\Console\Kernel中'Console\Kernel这一段转为目录，至于怎么转的我们在下面“Composer自动加载源码分析——运行”讲。核心类ClassLoader将loadClass()函数注册到PHP SPL中的spl_autoload_register()里面去，这个函数的来龙去脉我们之前文章讲过。这样，每当PHP遇到一个不认识的命名空间的时候，PHP会自动调用注册到spl_autoload_register里面的函数堆栈，运行其中的每个函数，直到找到命名空间对应的文件。

# Composer自动加载源码分析——全局函数的自动加载

Composer不止可以自动加载命名空间，还可以加载全局函数。怎么实现的呢？很简单，把全局函数写到特定的文件里面去，在程序运行前挨个require就行了。这个就是composer自动加载的第五步，加载全局函数。
```

```
  ```php
  if ($useStaticLoader) {
        $includeFiles = Composer\Autoload\ComposerStaticInit832e
a71bfb9a4128da8660baedaac82e::$files;
    } else {
        $includeFiles = require __DIR__ . '/autoload_files.php';
    }
    foreach ($includeFiles as $fileIdentifier => $file) {
        composerRequire832ea71bfb9a4128da8660baedaac82e($fileIde
ntifier, $file);
    }
    ```
```

跟核心类的初始化一样，全局函数自动加载也分为两种：静态初始化和普通初始化，静态加载只支持PHP5.6以上并且不支持HHVM。

静态初始化：

ComposerStaticInit832ea71bfb9a4128da8660baedaac82e::$files：

```
   ```php
   public static $files = array (
    '0e6d7bf4a5811bfa5cf40c5ccd6fae6a' => __DIR__ . '/..' . '/sy
mfony/polyfill-mbstring/bootstrap.php',
    '667aeda72477189d0494fecd327c3641' => __DIR__ . '/..' . '/sy
mfony/var-dumper/Resources/functions/dump.php',
    ...
);
```
```

看到这里我们可能又要有疑问了，为什么不直接放文件路径名，还要一个hash干什么呢？这个我们一会儿讲，我们这里先了解一下这个数组的结构。

普通初始化

autoload_files:

````
```php
$vendorDir = dirname(dirname(__FILE__));
$baseDir = dirname($vendorDir);

return array(
'0e6d7bf4a5811bfa5cf40c5ccd6fae6a' => $vendorDir . '/symfony/polyfill-mbstring/bootstrap.php',
'667aeda72477189d0494fecd327c3641' => $vendorDir . '/symfony/var-dumper/Resources/functions/dump.php',
   ....
);
```
````

其实跟静态初始化区别不大。

加载全局函数

````
```php
class ComposerAutoloaderInit832ea71bfb9a4128da8660baedaac82e{
  public static function getLoader(){
      ...
      foreach ($includeFiles as $fileIdentifier => $file) {
        composerRequire832ea71bfb9a4128da8660baedaac82e($fileIde
ntifier, $file);
      }
      ...
  }
}

function composerRequire832ea71bfb9a4128da8660baedaac82e($fileId
entifier, $file)
 {
    if (empty(\$GLOBALS['__composer_autoload_files'][\$fileIdent
ifier])) {
        require $file;

        $GLOBALS['__composer_autoload_files'][$fileIdentifier] =
 true;
    }
}
```
````

这一段很有讲究，第一个问题：为什么自动加载引导类的getLoader()函数不直接require $includeFiles里面的每个文件名，而要用类外面的函数composerRequire832ea71bfb9a4128da8660baedaac82e0？(顺便说下这个函数名hash仍然为了避免和用户定义函数冲突)因为怕有人在全局函数所在的文件写**$this**或者**self**。　　假如$includeFiles有个app/helper.php文件，这个helper.php文件的函数外有一行代码：$this->foo()，如果引导类在getLoader()函数直接require($file)，那么引导类就会运行这句代码，调用自己的foo()函数，这显然是错的。事实上helper.php就不应该出现$this或self这样的代码，这样写一般都是用户写错了的，一旦这样的事情发生，第一种情况：引导类恰好有foo()函数，那么就会莫名其妙执行了引导类的foo();第二种情况：引导类没有foo()函数，但是却甩出来引导类没有foo()方法这样的错误提示，用户不知道自己哪里错了。把require语句放到引导类的外面，遇到$this或者self，程序就会告诉用户根本没有类，$this或self无效，

错误信息更加明朗。　　第二个问题，为什么要用hash作为$fileIdentifier，上面的代码明显可以看出来这个变量是用来控制全局函数只被require一次的，那为什么不用require_once呢？事实上require_once比require效率低很多，使用全局变量$GLOBALS这样控制加载会更快。还有一个原因我猜测应该是require_once对相对路径的支持并不理想，所以composer尽量少用require_once。

Composer自动加载源码分析——运行

　　我们终于来到了核心的核心——composer自动加载的真相，命名空间如何通过composer转为对应目录文件的奥秘就在这一章。　　前面说过，ClassLoader的register()函数将loadClass()函数注册到PHP的SPL函数堆栈中，每当PHP遇到不认识的命名空间时就会调用函数堆栈的每个函数，直到加载命名空间成功。所以loadClass()函数就是自动加载的关键 了。 loadClass():

```php
public function loadClass($class)
{
    if ($file = $this->findFile($class)) {
        includeFile($file);

        return true;
    }
}

public function findFile($class)
{
    // work around for PHP 5.3.0 - 5.3.2 https://bugs.php.net/50
731
    if ('\\' == $class[0]) {
        $class = substr($class, 1);
    }

    // class map lookup
    if (isset($this->classMap[$class])) {
        return $this->classMap[$class];
    }
    if ($this->classMapAuthoritative) {
        return false;
```

```
    }

    $file = $this->findFileWithExtension($class, '.php');

    // Search for Hack files if we are running on HHVM
    if ($file === null && defined('HHVM_VERSION')) {
        $file = $this->findFileWithExtension($class, '.hh');
    }

    if ($file === null) {
        // Remember that this class does not exist.
        return $this->classMap[$class] = false;
    }

    return $file;
}
```

我们看到loadClass()，主要调用findFile()函数。findFile()在解析命名空间的时候主要分为两部分：classMap和findFileWithExtension()函数。classMap很简单，直接看命名空间是否在映射数组中即可。麻烦的是findFileWithExtension()函数，这个函数包含了PSR0和PSR4标准的实现。还有个值得我们注意的是查找路径成功后includeFile()仍然类外面的函数，并不是ClassLoader的成员函数，原理跟上面一样，防止有用户写$this或self。还有就是如果命名空间是以\开头的，要去掉\然后再匹配。 findFileWithExtension：

```php
     private function findFileWithExtension($class, $ext)
{
    // PSR-4 lookup
    $logicalPathPsr4 = strtr($class, '\\', DIRECTORY_SEPARATOR)
. $ext;

    $first = $class[0];
    if (isset($this->prefixLengthsPsr4[$first])) {
        foreach ($this->prefixLengthsPsr4[$first] as $prefix =>
$length) {
            if (0 === strpos($class, $prefix)) {
                foreach ($this->prefixDirsPsr4[$prefix] as $dir)
```

```
 {
                    if (file_exists($file = $dir . DIRECTORY_SEP
ARATOR . substr($logicalPathPsr4, $length))) {
                        return $file;
                    }
                }
            }
        }
    }

    // PSR-4 fallback dirs
    foreach ($this->fallbackDirsPsr4 as $dir) {
        if (file_exists($file = $dir . DIRECTORY_SEPARATOR . $lo
gicalPathPsr4)) {
            return $file;
        }
    }

    // PSR-0 lookup
    if (false !== $pos = strrpos($class, '\\')) {
        // namespaced class name
        $logicalPathPsr0 = substr($logicalPathPsr4, 0, $pos + 1)
            . strtr(substr($logicalPathPsr4, $pos + 1), '_', DIR
ECTORY_SEPARATOR);
    } else {
        // PEAR-like class name
        $logicalPathPsr0 = strtr($class, '_', DIRECTORY_SEPARATO
R) . $ext;
    }

    if (isset($this->prefixesPsr0[$first])) {
        foreach ($this->prefixesPsr0[$first] as $prefix => $dirs
) {
            if (0 === strpos($class, $prefix)) {
                foreach ($dirs as $dir) {
                    if (file_exists($file = $dir . DIRECTORY_SEP
ARATOR . $logicalPathPsr0)) {
                        return $file;
                    }
                }
```

```
            }
        }
    }

    // PSR-0 fallback dirs
    foreach ($this->fallbackDirsPsr0 as $dir) {
        if (file_exists($file = $dir . DIRECTORY_SEPARATOR . $lo
gicalPathPsr0)) {
            return $file;
        }
    }

    // PSR-0 include paths.
    if ($this->useIncludePath && $file = stream_resolve_include_
path($logicalPathPsr0)) {
        return $file;
    }
}
```
```

　　下面我们通过举例来说下上面代码的流程：　　如果我们在代码中写下'phpDocumentor\Reflection\example'，PHP会通过SPL调用loadClass->findFile->findFileWithExtension。首先默认用php作为文件后缀名调用findFileWithExtension函数里，利用PSR4标准尝试解析目录文件，如果文件不存在则继续用PSR0标准解析，如果解析出来的目录文件仍然不存在，但是环境是HHVM虚拟机，继续用后缀名为hh再次调用findFileWithExtension函数，如果不存在，说明此命名空间无法加载，放到classMap中设为false，以便以后更快地加载。

　　对于phpDocumentor\Reflection\example，当尝试利用PSR4标准映射目录时，步骤如下：

# PSR4标准加载

> - 将\转为文件分隔符/，加上后缀php或hh，得到$logicalPathPsr4即phpDocumentor//Reflection//example.php(hh);
> - 利用命名空间第一个字母p作为前缀索引搜索prefixLengthsPsr4数组，查到下面这个数组：
```

```
    ```php
    p' =>
        array (
            'phpDocumentor\\Reflection\\' => 25,
            'phpDocumentor\\Fake\\' => 19,
      )
    ```
```

> - 遍历这个数组，得到两个顶层命名空间phpDocumentor\Reflection\和phpDocumentor\Fake\
> - 用这两个顶层命名空间与phpDocumentor\Reflection\example_e相比较，可以得到phpDocumentor\Reflection\这个顶层命名空间
> - 在prefixLengthsPsr4映射数组中得到phpDocumentor\Reflection\长度为25。
> - 在prefixDirsPsr4映射数组中得到phpDocumentor\Reflection\的目录映射为：

```
```php
'phpDocumentor\\Reflection\\' =>
    array (
        0 => __DIR__ . '/..' . '/phpdocumentor/reflection-common
/src',
        1 => __DIR__ . '/..' . '/phpdocumentor/type-resolver/src
',
        2 => __DIR__ . '/..' . '/phpdocumentor/reflection-docblo
ck/src',
    ),
```
```

> - 遍历这个映射数组，得到三个目录映射；
> - 查看 "目录+文件分隔符//+substr($logicalPathPsr4, $length)"文件是否存在，存在即返回。这里就是'_DIR_/../phpdocumentor/reflection-common/src + /+ substr(phpDocumentor/Reflection/example_e.php(hh),25)'
> - 如果失败，则利用fallbackDirsPsr4数组里面的目录继续判断是否存在文件，具体方法是"目录+文件分隔符//+$logicalPathPsr4"

PSR0标准加载

如果PSR4标准加载失败，则要进行PSR0标准加载：

> - 找到phpDocumentor\Reflection\examplee最后"\"的位置，将其后面文件名中'"'字符转为文件分隔符"/",得到$logicalPathPsr0即phpDocumentor/Reflection/example/e.php(hh) 利用命名空间第一个字母p作为前缀索引搜索prefixLengthsPsr4数组，查到下面这个数组：

```
```php
'P' =>
    array (
        'Prophecy\\' =>
        array (
            0 => __DIR__ . '/..' . '/phpspec/prophecy/src',
        ),
        'phpDocumentor' =>
        array (
            0 => __DIR__ . '/..' . '/erusev/parsedown',
        ),
    ),
 ```
```

> - 遍历这个数组，得到两个顶层命名空间phpDocumentor和Prophecy
> - 用这两个顶层命名空间与phpDocumentor\Reflection\example_e相比较，可以得到phpDocumentor这个顶层命名空间
> - 在映射数组中得到phpDocumentor目录映射为'_DIR_ . '/..' . '/erusev/parsedown'
> - 查看“目录+文件分隔符//+$logicalPathPsr0”文件是否存在，存在即返回。这里就是“_DIR_ . '/..' . '/erusev/parsedown + //+ phpDocumentor//Reflection//example/e.php(hh)”
> - 如果失败，则利用fallbackDirsPsr0数组里面的目录继续判断是否存在文件，具体方法是“目录+文件分隔符//+$logicalPathPsr0”
> - 如果仍然找不到，则利用stream_resolve_include_path()，在当前include目录寻找该文件，如果找到返回绝对路径。

结语

经过三篇文章，终于写完了PHP Composer自动加载的原理与实现，结下来我们开始讲解laravel框架下的门面Facade,这个门面功能和自动加载有着一些联系.

前言

这篇文章我们开始讲laravel框架中的门面Facade，什么是门面呢？官方文档：

> Facades（读音：/fə'säd/ ）为应用程序的服务容器中可用的类提供了一个「静态」接口。Laravel 自带了很多 facades ，几乎可以用来访问到 Laravel 中所有的服务。Laravel facades 实际上是服务容器中那些底层类的「静态代理」，相比于传统的静态方法， facades 在提供了简洁且丰富的语法同时，还带来了更好的可测试性和扩展性。

什么意思呢？首先，我们要知道laravel框架的核心就是个Ioc容器即服务容器，功能类似于一个工厂模式，是个高级版的工厂。laravel的其他功能例如路由、缓存、日志、数据库其实都是类似于插件或者零件一样，叫做服务。Ioc容器主要的作用就是生产各种零件，就是提供各个服务。在laravel中，如果我们想要用某个服务，该怎么办呢？最简单的办法就是调用服务容器的make函数，或者利用依赖注入，或者就是今天要讲的门面Facade。门面相对于其他方法来说，最大的特点就是简洁，例如我们经常使用的Router，如果利用服务容器的make：

````
```php
App::make('router')->get('/', function () {
  return view('welcome');
});
```
````

如果利用门面：

````
```php
Route::get('/', function () {
  return view('welcome');
});
```
````

可以看出代码更加简洁。其实，下面我们就会介绍门面最后调用的函数也是服务容器的make函数。

Facade的原理

我们以Route为例，来讲解一下门面Facade的原理与实现。我们先来看Route的门面类：

```php
class Route extends Facade
{
    protected static function getFacadeAccessor()
    {
        return 'router';
    }
}
```

很简单吧？其实每个门面类也就是重定义一下getFacadeAccessor函数就行了，这个函数返回服务的唯一名称：router。需要注意的是要确保这个名称可以用服务容器的make函数创建成功(App::make('router'))，原因我们马上就会讲到。

那么当我们写出Route::get()这样的语句时，到底发生了什么呢？奥秘就在基类Facade中。

```php
public static function __callStatic($method, $args)
{
    $instance = static::getFacadeRoot();

    if (! $instance) {
        throw new RuntimeException('A facade root has not been s
et.');
    }

    return $instance->$method(...$args);
}
```

当运行Route::get()时，发现门面Route没有静态get()函数，PHP就会调用这个魔术函数__callStatic。我们看到这个魔术函数做了两件事：获得对象实例，利用对象调用get()函数。首先先看看如何获得对象实例的：

```php
public static function getFacadeRoot()
{
    return static::resolveFacadeInstance(static::getFacadeAccessor());
}
protected static function getFacadeAccessor()
{
    throw new RuntimeException('Facade does not implement getFacadeAccessor method.');
}
protected static function resolveFacadeInstance($name)
{
    if (is_object($name)) {
        return $name;
    }

    if (isset(static::$resolvedInstance[$name])) {
        return static::$resolvedInstance[$name];
    }

    return static::$resolvedInstance[$name] = static::$app[$name];
}
```

我们看到基类getFacadeRoot()调用了getFacadeAccessor()，也就是我们的服务重载的函数，如果调用了基类的getFacadeAccessor，就会抛出异常。在我们的例子里getFacadeAccessor()返回了“router”，接下来getFacadeRoot()又调用了resolveFacadeInstance()。在这个函数里重点就是

```php
return static::$resolvedInstance[$name] = static::$app[$name];
```

我们看到，在这里利用了$app也就是服务容器创建了“router”，创建成功后放入$resolvedInstance作为缓存，以便以后快速加载。　　好了，Facade的原理到这里就讲完了，但是到这里我们有个疑惑，为什么代码中写Route就可以调用Illuminate\Support\Facades\Route呢？这个就是别名的用途了，很多门面都有自己的别名，这样我们就不必在代码里面写use Illuminate\Support\Facades\Route，而是可以直接用Route了。

别名Aliases

为什么我们可以在laravel中全局用Route，而不需要使用use Illuminate\Support\Facades\Route?其实奥秘在于一个PHP函数：class_alias，它可以为任何类创建别名。laravel在启动的时候为各个门面类调用了class_alias函数，因此不必直接用类名，直接用别名即可。在config文件夹的app文件里面存放着门面与类名的映射：

````
```php
'aliases' => [

    'App' => Illuminate\Support\Facades\App::class,
    'Artisan' => Illuminate\Support\Facades\Artisan::class,
    'Auth' => Illuminate\Support\Facades\Auth::class,
    ...
    ]

```
````

下面我们来看看laravel是如何为门面类创建别名的。

启动别名Aliases服务

说到laravel的启动，我们离不开index.php：

~~~
```php
require __DIR__.'/../bootstrap/autoload.php';

$app = require_once __DIR__.'/../bootstrap/app.php';

$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);

$response = $kernel->handle(
$request = Illuminate\Http\Request::capture()
);
...
```
~~~

第一句就是我们前面博客说的composer的自动加载，接下来第二句获取laravel核心的Ioc容器，第三句“制造”出Http请求的内核，第四句是我们这里的关键，这句牵扯很大，laravel里面所有功能服务的注册加载，乃至Http请求的构造与传递都是这一句的功劳。

~~~
```php
$request = Illuminate\Http\Request::capture()
```
~~~

这句是laravel通过全局$_SERVER数组构造一个Http请求的语句，接下来会调用Http的内核函数handle：

````
```php
    public function handle($request)
    {
        try {
            $request->enableHttpMethodParameterOverride();

            $response = $this->sendRequestThroughRouter($request
);
        } catch (Exception $e) {
            $this->reportException($e);

            $response = $this->renderException($request, $e);
        } catch (Throwable $e) {
            $this->reportException($e = new FatalThrowableError(
$e));

            $response = $this->renderException($request, $e);
        }

        event(new Events\RequestHandled($request, $response));

        return $response;
    }
```
````

在handle函数方法中enableHttpMethodParameterOverride函数是允许在表单中使用delete、put等类型的请求。我们接着看sendRequestThroughRouter：

~~~~
```php
    protected function sendRequestThroughRouter($request)
    {
        $this->app->instance('request', $request);

        Facade::clearResolvedInstance('request');

        $this->bootstrap();

        return (new Pipeline($this->app))
                    ->send($request)
                    ->through($this->app->shouldSkipMiddleware()
 ? [] :
                            $this->middleware)
                    ->then($this->dispatchToRouter());
    }
```
~~~~

前两句是在laravel的Ioc容器设置request请求的对象实例，Facade中清楚request的缓存实例。bootstrap：

````
```php
    public function bootstrap()
    {
        if (! $this->app->hasBeenBootstrapped()) {
            $this->app->bootstrapWith($this->bootstrappers());
        }
    }

    protected $bootstrappers = [
        \Illuminate\Foundation\Bootstrap\LoadEnvironmentVariable
s::class,
        \Illuminate\Foundation\Bootstrap\LoadConfiguration::clas
s,
        \Illuminate\Foundation\Bootstrap\HandleExceptions::class
,
        \Illuminate\Foundation\Bootstrap\RegisterFacades::class,
        \Illuminate\Foundation\Bootstrap\RegisterProviders::clas
s,
        \Illuminate\Foundation\Bootstrap\BootProviders::class,
];
```
````

$bootstrappers是Http内核里专门用于启动的组件，bootstrap函数中调用Ioc容器的bootstrapWith函数来创建这些组件并利用组件进行启动服务。app->bootstrapWith：

```php
    public function bootstrapWith(array $bootstrappers)
    {
        $this->hasBeenBootstrapped = true;

        foreach ($bootstrappers as $bootstrapper) {
            $this['events']->fire('bootstrapping: '.$bootstrapper, [$this]);

            $this->make($bootstrapper)->bootstrap($this);

            $this['events']->fire('bootstrapped: '.$bootstrapper, [$this]);
        }
    }
```

可以看到bootstrapWith函数也就是利用Ioc容器创建各个启动服务的实例后，回调启动自己的函数bootstrap，在这里我们只看我们Facade的启动组件

```php
\Illuminate\Foundation\Bootstrap\RegisterFacades::class
```

RegisterFacades的bootstrap函数：

````
```php
class RegisterFacades
{
    public function bootstrap(Application $app)
    {
        Facade::clearResolvedInstances();

        Facade::setFacadeApplication($app);

        AliasLoader::getInstance($app->make('config')->get('app.
aliases', []))
                        ->register();
    }
}
```
````

可以看出来，bootstrap做了一下几件事：

> 1. 清除了Facade中的缓存
> 2. 设置Facade的Ioc容器
> 3. 获得我们前面讲的config文件夹里面app文件aliases别名映射数组
> 4. 使用aliases实例化初始化AliasLoader
> 5. 调用AliasLoader->register()

````
```php
    public function register()
    {
        if (! $this->registered) {
            $this->prependToLoaderStack();

            $this->registered = true;
        }
    }

    protected function prependToLoaderStack()
    {
        spl_autoload_register([$this, 'load'], true, true);
    }
```
````

我们可以看出，别名服务的启动关键就是这个spl_autoload_register，这个函数我们应该很熟悉了，在自动加载中这个函数用于解析命名空间，在这里用于解析别名的真正类名。

别名Aliases服务

我们首先来看看被注册到spl_autoload_register的函数，load：

````
```php
    public function load($alias)
    {
        if (static::$facadeNamespace && strpos($alias,
                                                static::$facadeNamespace
) === 0) {
            $this->loadFacade($alias);

            return true;
        }

        if (isset($this->aliases[$alias])) {
            return class_alias($this->aliases[$alias], $alias);
        }
    }
```
````

这个函数的下面很好理解，就是class_alias利用别名映射数组将别名映射到真正的门面类中去，但是上面这个是什么呢?实际上，这个是laravel5.4版本新出的功能叫做实时门面服务。

实时门面服务

其实门面功能已经很简单了，我们只需要定义一个类继承Facade即可，但是laravel5.4打算更近一步——自动生成门面子类，这就是实时门面。　　实时门面怎么用？看下面的例子：

````
```php
namespace App\Services;

class PaymentGateway
{
    protected $tax;

    public function __construct(TaxCalculator $tax)
    {
        $this->tax = $tax;
    }
}
```
````

这是一个自定义的类，如果我们想要为这个类定义一个门面，在laravel5.4我们可以这么做：

````
```php
use Facades\ {
    App\Services\PaymentGateway
};

Route::get('/pay/{amount}', function ($amount) {
    PaymentGateway::pay($amount);
});
```
````

当然如果你愿意，你还可以在alias数组为门面添加一个别名映射"PaymentGateway" => "use Facades\App\Services\PaymentGateway"，这样就不用写这么长的名字了。　　那么这么做的原理是什么呢？我们接着看源码：

````
```php
protected static $facadeNamespace = 'Facades\\';
if (static::$facadeNamespace && strpos($alias, static::$facadeNa
mespace) === 0) {
   $this->loadFacade($alias);

   return true;
}
```
````

如果命名空间是以Facades\开头的，那么就会调用实时门面的功能，调用loadFacade函数：

````
```php
protected function loadFacade($alias)
{
    tap($this->ensureFacadeExists($alias), function ($path) {
        require $path;
    });
}
```
````

tap是laravel的全局帮助函数，ensureFacadeExists函数负责自动生成门面类，loadFacade负责加载门面类：

~~~
   ```php
    protected function ensureFacadeExists($alias)
    {
        if (file_exists($path = storage_path('framework/cache/fa
cade-'.sha1($alias).'.php'))) {
            return $path;
        }

        file_put_contents($path, $this->formatFacadeStub(
            $alias, file_get_contents(__DIR__.'/stubs/facade.stu
b')
        ));

        return $path;
    }
    ```
~~~

可以看出来，laravel框架生成的门面类会放到stroge/framework/cache/文件夹下，名字以facade开头，以命名空间的哈希结尾。如果存在这个文件就会返回，否则就要利用file_put_contents生成这个文件，formatFacadeStub：

~~~
    ```php
    protected function formatFacadeStub($alias, $stub)
    {
        $replacements = [
            str_replace('/', '\\', dirname(str_replace('\\', '/'
, $alias))),
            class_basename($alias),
            substr($alias, strlen(static::$facadeNamespace)),
        ];

        return str_replace(
            ['DummyNamespace', 'DummyClass', 'DummyTarget'], $re
placements, $stub
        );
    }
```
~~~

简单的说，对于Facades\App\Services\PaymentGateway，$replacements第一项是门面命名空间，将Facades\App\Services\PaymentGateway转为Facades/App/Services/PaymentGateway，取前面Facades/App/Services/，再转为命名空间Facades\App\Services\；第二项是门面类名，PaymentGateway；第三项是门面类的服务对象，App\Services\PaymentGateway，用这些来替换门面的模板文件：

````
```php
<?php

namespace DummyNamespace;

use Illuminate\Support\Facades\Facade;

/**
 * @see \DummyTarget
 */
class DummyClass extends Facade
{
    /**
     * Get the registered name of the component.
     *
     * @return string
     */
    protected static function getFacadeAccessor()
    {
        return 'DummyTarget';
    }
}
```
````

替换后的文件是：

````
```php
<?php

namespace Facades\App\Services\;

use Illuminate\Support\Facades\Facade;

/**
 * @see \DummyTarget
 */
class PaymentGateway extends Facade
{
    /**
     * Get the registered name of the component.
     *
     * @return string
     */
    protected static function getFacadeAccessor()
    {
        return 'App\Services\PaymentGateway';
    }
}
```
````

就是这么简单！！！

结语

门面的原理就是这些，相对来说门面服务的原理比较简单，和自动加载相互配合使得代码更加简洁，希望大家可以更好的使用这些门面！

服务容器

在说 Ioc 容器之前，我们需要了解什么是 Ioc 容器。

> Laravel 服务容器是一个用于管理类依赖和执行依赖注入的强大工具。

在理解这句话之前，我们需要先了解一下服务容器的来龙去脉：laravel神奇的服务容器。这篇博客告诉我们，服务容器就是工厂模式的升级版，对于传统的工厂模式来说，虽然解耦了对象和外部资源之间的关系，但是工厂和外部资源之间却存在了耦和。而服务容器在为对象创建了外部资源的同时，又与外部资源没有任何关系，这个就是 Ioc 容器。　所谓的依赖注入和控制反转: 依赖注入和控制反转，就是

> 只要不是由内部生产（比如初始化、构造函数 __construct 中通过工厂方法、自行手动 new 的），而是由外部以参数或其他形式注入的，都属于依赖注入（DI）

也就是说：

> 依赖注入是从应用程序的角度在描述，可以把依赖注入描述完整点：应用程序依赖容器创建并注入它所需要的外部资源；
>
> 控制反转是从容器的角度在描述，描述完整点：容器控制应用程序，由容器反向的向应用程序注入应用程序所需要的外部资源。

Laravel中的服务容器

Laravel服务容器主要承担两个作用：绑定与解析。

绑定

所谓的绑定就是将接口与实现建立对应关系。几乎所有的服务容器绑定都是在服务提供者中完成，也就是在服务提供者中绑定。

> 如果一个类没有基于任何接口那么就没有必要将其绑定到容器。容器并不需要被告知如何构建对象，因为它会使用 PHP 的反射服务自动解析出具体的对象。

也就是说，如果需要依赖注入的外部资源如果没有接口，那么就不需要绑定，直接利用服务容器进行解析就可以了，服务容器会根据类名利用反射对其进行自动构造。

bind绑定

绑定有多种方法，首先最常用的是bind函数的绑定：

- 绑定自身

```
$this->app->bind('App\Services\RedisEventPusher', null);
```

- 绑定闭包

```
$this->app->bind('name', function () {
    return 'Taylor';
});//闭包返回变量

$this->app->bind('HelpSpot\API', function () {
    return HelpSpot\API::class;
});//闭包直接提供类实现方式

public function testSharedClosureResolution()
{
    $container = new Container;
    $class = new stdClass;
    $container->bind('class', function () use ($class) {
        return $class;
    });

    $this->assertSame($class, $container->make('class'));
}//闭包返回类变量

$this->app->bind('HelpSpot\API', function () {
    return new HelpSpot\API();
});//闭包直接提供类实现方式

$this->app->bind('HelpSpot\API', function ($app) {
    return new HelpSpot\API($app->make('HttpClient'));
});//闭包返回需要依赖注入的类
```

- 绑定接口

```
public function testCanBuildWithoutParameterStackWithConstructors
()
{
    $container = new Container;
    $container->bind('Illuminate\Tests\Container\IContainerContr
actStub',
                     'Illuminate\Tests\Container\ContainerImplem
entationStub');

    $this->assertInstanceOf(ContainerDependentStub::class,
                            $container->build(ContainerDependent
Stub::class));
}

interface IContainerContractStub
{
}

class ContainerImplementationStub implements IContainerContractS
tub
{
}

class ContainerDependentStub
{
    public $impl;
    public function __construct(IContainerContractStub $impl)
    {
        $this->impl = $impl;
    }
}
```

这三种绑定方式中，第一种绑定自身一般用于绑定单例。

bindif绑定

```
public function testBindIfDoesntRegisterIfServiceAlreadyRegister
ed()
{
    $container = new Container;
    $container->bind('name', function ()
         return 'Taylor';
     });

    $container->bindIf('name', function () {
         return 'Dayle';
    });

    $this->assertEquals('Taylor', $container->make('name'));
}
```

singleton绑定

singleton 方法绑定一个只需要解析一次的类或接口到容器，然后接下来对容器的调用将会返回同一个实例：

```
$this->app->singleton('HelpSpot\API', function ($app) {
    return new HelpSpot\API($app->make('HttpClient'));
});
```

值得注意的是，singleton绑定在解析的时候若存在参数重载，那么就自动取消单例模式。

```
public function testSingletonBindingsNotRespectedWithMakeParamet
ers()
{
    $container = new Container;

    $container->singleton('foo', function ($app, $config) {
        return $config;
    });

    $this->assertEquals(['name' => 'taylor'], $container->makeWi
th('foo', ['name' => 'taylor']));
    $this->assertEquals(['name' => 'abigail'], $container->makeW
ith('foo', ['name' => 'abigail']));
    }
```

instance绑定

我们还可以使用 instance 方法绑定一个已存在的对象实例到容器，随后调用容器将总是返回给定的实例：

```
$api = new HelpSpot\API(new HttpClient);
$this->app->instance('HelpSpot\Api', $api);
```

Context绑定

有时侯我们可能有两个类使用同一个接口，但我们希望在每个类中注入不同实现，例如，两个控制器依赖 Illuminate\Contracts\Filesystem\Filesystem 契约的不同实现。Laravel 为此定义了简单、平滑的接口：

```
use Illuminate\Support\Facades\Storage;
use App\Http\Controllers\VideoController;
use App\Http\Controllers\PhotoControllers;
use Illuminate\Contracts\Filesystem\Filesystem;

$this->app->when(StorageController::class)
          ->needs(Filesystem::class)
          ->give(function () {
            Storage::class
          });//提供类名

$this->app->when(PhotoController::class)
          ->needs(Filesystem::class)
          ->give(function () {
             return new Storage();
          });//提供实现方式

$this->app->when(VideoController::class)
          ->needs(Filesystem::class)
          ->give(function () {
            return new Storage($app->make(Disk::class));
          });//需要依赖注入
```

原始值绑定

我们可能有一个接收注入类的类，同时需要注入一个原生的数值比如整型，可以结合上下文轻松注入这个类需要的任何值：

```
$this->app->when('App\Http\Controllers\UserController')
          ->needs('$variableName')
          ->give($value);
```

数组绑定

数组绑定一般用于绑定闭包和变量，但是不能绑定接口，否则只能返回接口的实现类名字符串,并不能返回实现类的对象。

```
public function testArrayAccess()
{
    $container = new Container;
    $container[IContainerContractStub::class] = ContainerImpleme
ntationStub::class;

    $this->assertTrue(isset($container[IContainerContractStub::c
lass]));
    $this->assertEquals(ContainerImplementationStub::class,
                        $container[IContainerContractStub::class
]);

    unset($container['something']);
    $this->assertFalse(isset($container['something']));
}
```

标签绑定

少数情况下，我们需要解析特定分类下的所有绑定，例如，你正在构建一个接收多个不同 Report 接口实现的报告聚合器，在注册完 Report 实现之后，可以通过 tag 方法给它们分配一个标签：

```
$this->app->bind('SpeedReport', function () {
  //
});

$this->app->bind('MemoryReport', function () {
  //
});

$this->app->tag(['SpeedReport', 'MemoryReport'], 'reports');
```

这些服务被打上标签后，可以通过 tagged 方法来轻松解析它们：

```
$this->app->bind('ReportAggregator', function ($app) {
    return new ReportAggregator($app->tagged('reports'));
});
```

```
  public function testContainerTags()
  {
          $container = new Container;
          $container->tag('Illuminate\Tests\Container\ContainerImplementationStub', 'foo', 'bar');
          $container->tag('Illuminate\Tests\Container\ContainerImplementationStubTwo', ['foo']);

          $this->assertCount(1, $container->tagged('bar'));
          $this->assertCount(2, $container->tagged('foo'));
          $this->assertInstanceOf('Illuminate\Tests\Container\ContainerImplementationStub', $container->tagged('foo')[0]);
          $this->assertInstanceOf('Illuminate\Tests\Container\ContainerImplementationStub', $container->tagged('bar')[0]);
          $this->assertInstanceOf('Illuminate\Tests\Container\ContainerImplementationStubTwo', $container->tagged('foo')[1]);

          $container = new Container;
          $container->tag(['Illuminate\Tests\Container\ContainerImplementationStub', 'Illuminate\Tests\Container\ContainerImplementationStubTwo'], ['foo']);
          $this->assertCount(2, $container->tagged('foo'));
          $this->assertInstanceOf('Illuminate\Tests\Container\ContainerImplementationStub', $container->tagged('foo')[0]);
          $this->assertInstanceOf('Illuminate\Tests\Container\ContainerImplementationStubTwo', $container->tagged('foo')[1]);

           $this->assertEmpty($container->tagged('this_tag_does_not_exist'));
  }
```

extend扩展

extend是在当原来的类被注册或者实例化出来后，可以对其进行扩展，而且可以支持多重扩展：

```
public function testExtendInstancesArePreserved()
{
    $container = new Container;
    $container->bind('foo', function () {
        $obj = new StdClass;
        $obj->foo = 'bar';

        return $obj;
    });

    $obj = new StdClass;
    $obj->foo = 'foo';
    $container->instance('foo', $obj);

    $container->extend('foo', function ($obj, $container) {
        $obj->bar = 'baz';
        return $obj;
    });

    $container->extend('foo', function ($obj, $container) {
        $obj->baz = 'foo';
        return $obj;
    });

    $this->assertEquals('foo', $container->make('foo')->foo);
    $this->assertEquals('baz', $container->make('foo')->bar);
    $this->assertEquals('foo', $container->make('foo')->baz);
}
```

Rebounds与Rebinding

绑定是针对接口的，是为接口提供实现方式的方法。我们可以对接口在不同的时间段里提供不同的实现方法，一般来说，对同一个接口提供新的实现方法后，不会对已经实例化的对象产生任何影响。但是在一些场景下，在提供新的接口实现后，我们希望对已经实例化的对象重新做一些改变，这个就是 rebinding 函数的用途。 下面就是一个例子：

```
abstract class Car
{
    public function __construct(Fuel $fuel)
    {
        $this->fuel = $fuel;
    }

    public function refuel($litres)
    {
        return $litres * $this->fuel->getPrice();
    }

    public function setFuel(Fuel $fuel)
    {
        $this->fuel = $fuel;
    }

}

class JeepWrangler extends Car
{
  //
}

interface Fuel
{
    public function getPrice();
}

class Petrol implements Fuel
{
    public function getPrice()
    {
        return 130.7;
    }
}
```

我们在服务容器中是这样对car接口和fuel接口绑定的：

```
$this->app->bind('fuel', function ($app) {
    return new Petrol;
});

$this->app->bind('car', function ($app) {
    return new JeepWrangler($app['fuel']);
});

$this->app->make('car');
```

如果car被服务容器解析实例化成对象之后，有人修改了 fuel 接口的实现，从 Petrol 改为 PremiumPetrol：

```
$this->app->bind('fuel', function ($app) {
    return new PremiumPetrol;
});
```

由于 car 已经被实例化，那么这个接口实现的改变并不会影响到 car 的实现，假若我们想要 car 的成员变量 fuel 随着 fuel 接口的变化而变化，我们就需要一个回调函数，每当对 fuel 接口实现进行改变的时候，都要对 car 的 fuel 变量进行更新，这就是 rebinding 的用途：

```
$this->app->bindShared('car', function ($app) {
    return new JeepWrangler($app->rebinding('fuel', function ($a
pp, $fuel) {
        $app['car']->setFuel($fuel);
    }));
});
```

服务别名

什么是服务别名

在说服务容器的解析之前，需要先说说服务的别名。什么是服务别名呢？不同于上一个博客中提到的 Facade 门面的别名(在 config/app 中定义)，这里的别名服务绑定名称的别名。通过服务绑定的别名，在解析服务的时候，跟不使用别名的效果一致。别名的作用也是为了同时支持全类型的服务绑定名称以及简短的服务绑定名称考虑的。　通俗的讲，假如我们想要创建 auth 服务，我们既可以这样写：

```
$this->app->make('auth')
```

又可以写成：

```
$this->app->make('\Illuminate\Auth\AuthManager::class')
```

还可以写成

```
$this->app->make('\Illuminate\Contracts\Auth\Factory::class')
```

后面两个服务的名字都是 auth 的别名，使用别名和使用 auth 的效果是相同的。

服务别名的递归

需要注意的是别名是可以递归的：

```
app()->alias('service', 'alias_a');
app()->alias('alias_a', 'alias_b');
app()-alias('alias_b', 'alias_c');
```

会得到：

```
'alias_a' => 'service'
'alias_b' => 'alias_a'
'alias_c' => 'alias_b'
```

服务别名的实现

那么这些别名是如何加载到服务容器里面的呢？实际上，服务容器里面有个 aliases 数组：

```
$aliases = [
  'app' => [\Illuminate\Foundation\Application::class, \Illuminate\Contracts\Container\Container::class, \Illuminate\Contracts\Foundation\Application::class],
  'auth' => [\Illuminate\Auth\AuthManager::class, \Illuminate\Contracts\Auth\Factory::class],
  'auth.driver' => [\Illuminate\Contracts\Auth\Guard::class],
  'blade.compiler' => [\Illuminate\View\Compilers\BladeCompiler::class],
  'cache' => [\Illuminate\Cache\CacheManager::class, \Illuminate\Contracts\Cache\Factory::class],
...
]
```

而服务容器的初始化的过程中，会运行一个函数：

```
public function registerCoreContainerAliases()
{
  foreach ($aliases as $key => $aliases) {
    foreach ($aliases as $alias) {
      $this->alias($key, $alias);
    }
  }
}

public function alias($abstract, $alias)
{
  $this->aliases[$alias] = $abstract;

  $this->abstractAliases[$abstract][] = $alias;
}
```

加载后，服务容器的aliases和abstractAliases数组：

```
$aliases = [
  'Illuminate\Foundation\Application' = "app"
  'Illuminate\Contracts\Container\Container' = "app"
  'Illuminate\Contracts\Foundation\Application' = "app"
  'Illuminate\Auth\AuthManager' = "auth"
  'Illuminate\Contracts\Auth\Factory' = "auth"
  'Illuminate\Contracts\Auth\Guard' = "auth.driver"
  'Illuminate\View\Compilers\BladeCompiler' = "blade.compiler"
  'Illuminate\Cache\CacheManager' = "cache"
  'Illuminate\Contracts\Cache\Factory' = "cache"
  ...
]
$abstractAliases = [
  app = {array} [3]
  0 = "Illuminate\Foundation\Application"
  1 = "Illuminate\Contracts\Container\Container"
  2 = "Illuminate\Contracts\Foundation\Application"
  auth = {array} [2]
  0 = "Illuminate\Auth\AuthManager"
  1 = "Illuminate\Contracts\Auth\Factory"
  auth.driver = {array} [1]
  0 = "Illuminate\Contracts\Auth\Guard"
  blade.compiler = {array} [1]
  0 = "Illuminate\View\Compilers\BladeCompiler"
  cache = {array} [2]
  0 = "Illuminate\Cache\CacheManager"
  1 = "Illuminate\Contracts\Cache\Factory"
  ...
]
```

服务解析

make 解析

有很多方式可以从容器中解析对象，首先，你可以使用 make 方法，该方法接收你想要解析的类名或接口名作为参数：

```
public function testAutoConcreteResolution()
{
    $container = new Container;
    $this->assertInstanceOf('Illuminate\Tests\Container\Containe
rConcreteStub',
           $container->make('Illuminate\Tests\Container\Containe
rConcreteStub'));
}

//带有依赖注入和默认值的解析
public function testResolutionOfDefaultParameters()
{
       $container = new Container;
       $instance = $container->make('Illuminate\Tests\Container\
ContainerDefaultValueStub');
       $this->assertInstanceOf('Illuminate\Tests\Container\Conta
inerConcreteStub',
                                 $instance->stub);
       $this->assertEquals('taylor', $instance->default);
}

//
public function testResolvingWithArrayOfParameters()
{
    $container = new Container;

    $instance = $container->makeWith(ContainerDefaultValueStub::
class, ['default' => 'adam']);
    $this->assertEquals('adam', $instance->default);

    $instance = $container->make(ContainerDefaultValueStub::clas
s);
    $this->assertEquals('taylor', $instance->default);

    $container->bind('foo', function ($app, $config) {
        return $config;
    });
    $this->assertEquals([1, 2, 3], $container->makeWith('foo', [1
, 2, 3]));
```

```
}

public function testNestedDependencyResolution()
{
    $container = new Container;
    $container->bind('Illuminate\Tests\Container\IContainerContr
actStub', 'Illuminate\Tests\Container\ContainerImplementationStu
b');
    $class = $container->make('Illuminate\Tests\Container\Contai
nerNestedDependentStub');
    $this->assertInstanceOf('Illuminate\Tests\Container\Containe
rDependentStub', $class->inner);
    $this->assertInstanceOf('Illuminate\Tests\Container\Containe
rImplementationStub', $class->inner->impl);
    }

class ContainerDefaultValueStub
{
    public $stub;
    public $default;
    public function __construct(ContainerConcreteStub $stub, $de
fault = 'taylor')
    {
        $this->stub = $stub;
        $this->default = $default;
    }
}

class ContainerConcreteStub
{
}

class ContainerImplementationStub implements IContainerContractS
tub
{
}

class ContainerDependentStub
{
    public $impl;
```

```
    public function __construct(IContainerContractStub $impl)
    {
        $this->impl = $impl;
    }
}
class ContainerNestedDependentStub
{
    public $inner;
    public function __construct(ContainerDependentStub $inner)
    {
        $this->inner = $inner;
    }
}
```

如果你所在的代码位置访问不了 $app 变量，可以使用辅助函数resolve：

```
$api = resolve('HelpSpot\API');
```

自动注入

```
namespace App\Http\Controllers;

use App\Users\Repository as UserRepository;

class UserController extends Controller{
  /**
  * 用户仓库实例
  */
  protected $users;

  /**
  * 创建一个控制器实例
  *
  * @param UserRepository $users 自动注入
  * @return void
  */
  public function __construct(UserRepository $users)
  {
    $this->users = $users;
  }
}
```

call 方法注入

make 解析是服务容器进行解析构建类对象时所用的方法，在实际应用中，还有另外一个需求，那就是当前已经获取了一个类对象，我们想要调用它的一个方法函数，这时发现这个方法中参数众多，如果一个个的 make 会比较繁琐，这个时候就要用到 call 解析了。我们可以看这个例子：

```
class TaskRepository{

    public function testContainerCall(User $user,Task $task){
        $this->assertInstanceOf(User::class, $user);

        $this->assertInstanceOf(Task::class, $task);
    }

    public static function testContainerCallStatic(User $user,Ta
sk $task){
        $this->assertInstanceOf(User::class, $user);

        $this->assertInstanceOf(Task::class, $task);
    }

    public function testCallback(){
        echo 'call callback successfully!';
    }

    public function testDefaultMethod(){
        echo 'default Method successfully!';
    }
}
```

闭包函数注入

```
    public function testCallWithDependencies()
    {
        $container = new Container;
        $result = $container->call(function (StdClass $foo, $bar = []) {
            return func_get_args();
        });

        $this->assertInstanceOf('stdClass', $result[0]);
        $this->assertEquals([], $result[1]);

        $result = $container->call(function (StdClass $foo, $bar = []) {
            return func_get_args();
        }, ['bar' => 'taylor']);

        $this->assertInstanceOf('stdClass', $result[0]);
        $this->assertEquals('taylor', $result[1]);
}
```

普通函数注入

```
public function testCallWithGlobalMethodName()
{
    $container = new Container;
    $result = $container->call('Illuminate\Tests\Container\containerTestInject');
    $this->assertInstanceOf('Illuminate\Tests\Container\ContainerConcreteStub', $result[0]);
    $this->assertEquals('taylor', $result[1]);
}
```

静态方法注入

服务容器的 call 解析主要依靠 call_user_func_array() 函数，关于这个函数可以查看 Laravel学习笔记之Callback Type - 来生做个漫画家，这个函数对类中的静态函数和非静态函数有一些区别，对于静态函数来说：

```
class ContainerCallTest
{
    public function testContainerCallStatic(){
        App::call(TaskRepository::class.'@testContainerCallStati
c');
        App::call(TaskRepository::class.'::testContainerCallStat
ic');
        App::call([TaskRepository::class,'testContainerCallStati
c']);
    }
}
```

服务容器调用类的静态方法有三种，注意第三种使用数组的形式，数组中可以直接传类名 TaskRepository::class；

非静态方法注入

对于类的非静态方法：

```
class ContainerCallTest
{
    public function testContainerCall(){
        $taskRepo = new TaskRepository();
        App::call(TaskRepository::class.'@testContainerCall');
        App::call([$taskRepo,'testContainerCall']);
    }
}
```

我们可以看到非静态方法只有两种调用方式，而且第二种数组传递的参数是类对象，原因就是 call_user_func_array函数的限制，对于非静态方法只能传递对象。

bindmethod 方法绑定

服务容器还有一个 bindmethod 的方法，可以绑定类的一个方法到自定义的函数：

```
public function testContainCallMethodBind(){

    App::bindMethod(TaskRepository::class.'@testContainerCallSta
tic',function () {
         $taskRepo = new TaskRepository();
         $taskRepo->testCallback();
    });

    App::call(TaskRepository::class.'@testContainerCallStatic');
    App::call(TaskRepository::class.'::testContainerCallStatic')
;
    App::call([TaskRepository::class,'testContainerCallStatic'])
;

    App::bindMethod(TaskRepository::class.'@testContainerCall',f
unction (TaskRepository $taskRepo) { $taskRepo->testCallback();
});

    $taskRepo = new TaskRepository();
    App::call(TaskRepository::class.'@testContainerCall');
    App::call([$taskRepo,'testContainerCall']);
}
```

从结果上看，bindmethod 不会对静态的第二种解析方法（ :: 解析方式）起作用，对于其他方式都会调用绑定的函数。

```
  public function testCallWithBoundMethod()
  {
      $container = new Container;
      $container->bindMethod('Illuminate\Tests\Container\Contain
erTestCallStub@unresolvable', function ($stub) {
          return $stub->unresolvable('foo', 'bar');
      });
      $result = $container->call('Illuminate\Tests\Container\Con
tainerTestCallStub@unresolvable');
      $this->assertEquals(['foo', 'bar'], $result);

      $container = new Container;
      $container->bindMethod('Illuminate\Tests\Container\Contain
erTestCallStub@unresolvable', function ($stub) {
          return $stub->unresolvable('foo', 'bar');
  });
      $result = $container->call([new ContainerTestCallStub, 'un
resolvable']);
      $this->assertEquals(['foo', 'bar'], $result);
  }

class ContainerTestCallStub
{
    public function unresolvable($foo, $bar)
    {
        return func_get_args();
    }
}
```

默认函数注入

```
public function testContainCallDefultMethod(){

    App::call(TaskRepository::class,[],'testContainerCall');

    App::call(TaskRepository::class,[],'testContainerCallStatic'
);

    App::bindMethod(TaskRepository::class.'@testContainerCallSta
tic',function () {
        $taskRepo = new TaskRepository();
        $taskRepo->testCallback();
    });

    App::bindMethod(TaskRepository::class.'@testContainerCall',f
unction (TaskRepository $taskRepo) {  $taskRepo->testCallback();
 });

    App::call(TaskRepository::class,[],'testContainerCall');

    App::call(TaskRepository::class,[],'testContainerCallStatic'
);

}
```

值得注意的是，这种默认函数注入的方法使得非静态的方法也可以利用类名去调用，并不需要对象。默认函数注入也回受到 bindmethod 函数的影响。

数组解析

```
app()['service'];
```

app($service)的形式

```
app('service');
```

服务容器事件

每当服务容器解析一个对象时就会触发一个事件。你可以使用 resolving 方法监听这个事件：

```
$this->app->resolving(function ($object, $app) {
  // 解析任何类型的对象时都会调用该方法...
});
$this->app->resolving(HelpSpot\API::class, function ($api, $app)
{
  // 解析「HelpSpot\API」类型的对象时调用...
});
$this->app->afterResolving(function ($object, $app) {
  // 解析任何类型的对象后都会调用该方法...
});
$this->app->afterResolving(HelpSpot\API::class, function ($api,
$app) {
  // 解析「HelpSpot\API」类型的对象后调用...
});
```

服务容器每次解析对象的时候，都会调用这些通过 resolving 和 afterResolving 函数传入的闭包函数，也就是触发这些事件。 注意：如果是单例，则只在解析时会触发一次

```
public function testResolvingCallbacksAreCalled()
{
    $container = new Container;
    $container->resolving(function ($object) {
        return $object->name = 'taylor';
    });
    $container->bind('foo', function () {
        return new StdClass;
    });
    $instance = $container->make('foo');

    $this->assertEquals('taylor', $instance->name);
}
```

```
public function testResolvingCallbacksAreCalledForType()
{
    $container = new Container;
    $container->resolving('StdClass', function ($object) {
        return $object->name = 'taylor';
    });
    $container->bind('foo', function () {
          return new StdClass;
    });
    $instance = $container->make('foo');

    $this->assertEquals('taylor', $instance->name);
}
public function testResolvingCallbacksShouldBeFiredWhenCalledWit
hAliases()
{
    $container = new Container;
    $container->alias('StdClass', 'std');
    $container->resolving('std', function ($object) {
        return $object->name = 'taylor';
    });
    $container->bind('foo', function () {
        return new StdClass;
    });
    $instance = $container->make('foo');

    $this->assertEquals('taylor', $instance->name);
}
```

装饰函数

容器的装饰函数有两种，wrap用于装饰call，factory用于装饰make：

```
public function testContainerWrap()
{
        $result = $container->wrap(function (StdClass $foo, $bar =
 []) {
            return func_get_args();
        }, ['bar' => 'taylor']);

        $this->assertInstanceOf('Closure', $result);
        $result = $result();

        $this->assertInstanceOf('stdClass', $result[0]);
        $this->assertEquals('taylor', $result[1]);
  }

public function testContainerGetFactory()
{
    $container = new Container;
    $container->bind('name', function () {
        return 'Taylor';
    });
    $factory = $container->factory('name');
    $this->assertEquals($container->make('name'), $factory());
}
```

容器重置flush

容器的重置函数flush会清空容器内部的aliases、abstractAliases、resolved、bindings、instances

```
public function testContainerFlushFlushesAllBindingsAliasesAndRe
solvedInstances()
{
    $container = new Container;
    $container->bind('ConcreteStub', function () {
        return new ContainerConcreteStub;
    }, true);
    $container->alias('ConcreteStub', 'ContainerConcreteStub');

    $concreteStubInstance = $container->make('ConcreteStub');
    $this->assertTrue($container->resolved('ConcreteStub'));
    $this->assertTrue($container->isAlias('ContainerConcreteStub'
));
    $this->assertArrayHasKey('ConcreteStub', $container->getBind
ings());
    $this->assertTrue($container->isShared('ConcreteStub'));

    $container->flush();
    $this->assertFalse($container->resolved('ConcreteStub'));
    $this->assertFalse($container->isAlias('ContainerConcreteStu
b'));
    $this->assertEmpty($container->getBindings());
    $this->assertFalse($container->isShared('ConcreteStub'));
}
```

> Written with StackEdit.

前言

在前面几个博客中，我详细讲了 Ioc 容器各个功能的使用、绑定的源码、解析的源码，今天这篇博客会详细介绍 Ioc 容器的一些细节，一些特性，以便更好地掌握容器的功能。

注：本文使用的测试类与测试对象都取自 laravel 的单元测试文件 src/illuminate/tests/Container/ContainerTest.php

rebind绑定特性

rebind 在绑定之前

instance 和 普通 bind 绑定一样，当重新绑定的时候都会调用 rebind 回调函数，但是有趣的是，对于普通 bind 绑定来说，rebind 回调函数被调用的条件是当前接口被解析过：

```
public function testReboundListeners()
{
    unset($_SERVER['__test.rebind']);

    $container = new Container;
    $container->rebinding('foo', function () {
        $_SERVER['__test.rebind'] = true;
    });
    $container->bind('foo', function () {
    });
    $container->make('foo');
    $container->bind('foo', function () {
    });

    $this->assertTrue($_SERVER['__test.rebind']);
}
```

所以遇到下面这样的情况,rebinding 的回调函数是不会调用的：

```
public function testReboundListeners()
{
    unset($_SERVER['__test.rebind']);

    $container = new Container;
    $container->rebinding('foo', function () {
        $_SERVER['__test.rebind'] = true;
    });
    $container->bind('foo', function () {
    });
    $container->bind('foo', function () {
    });

    $this->assertFalse(isset($_SERVER['__test.rebind']));
}
```

有趣的是对于 instance 绑定：

```
public function testReboundListeners()
{
    unset($_SERVER['__test.rebind']);

    $container = new Container;
    $container->rebinding('foo', function () {
        $_SERVER['__test.rebind'] = true;
    });
    $container->bind('foo', function () {
    });
    $container->instance('foo', function () {
    });

    $this->assertTrue(isset($_SERVER['__test.rebind']));
}
```

rebinding 回调函数却是可以被调用的。其实原因就是 instance 源码中 rebinding 回调函数调用的条件是 rebound 为真，而普通 bind 函数调用 rebinding 回调函数的条件是 resolved 为真. 目前笔者不是很清楚为什么要对 instance 和 bind 区别对待，希望有大牛指导。

rebind 在绑定之后

为了使得 rebind 回调函数在下一次的绑定中被激活，在 rebind 函数的源码中，如果判断当前对象已经绑定过，那么将会立即解析：

```
public function rebinding($abstract, Closure $callback)
{
    $this->reboundCallbacks[$abstract = $this->getAlias($abstrac
t)][] = $callback;

    if ($this->bound($abstract)) {
        return $this->make($abstract);
    }
}
```

单元测试代码：

```
public function testReboundListeners1()
{
    unset($_SERVER['__test.rebind']);

    $container = new Container;
    $container->bind('foo', function () {
        return 'foo';
    });

    $container->resolving('foo', function () {
        $_SERVER['__test.rebind'] = true;
    });

    $container->rebinding('foo', function ($container,$object) {
//会立即解析
        $container['foobar'] = $object.'bar';
    });

    $this->assertTrue($_SERVER['__test.rebind']);

    $container->bind('foo', function () {
    });

    $this->assertEquals('bar', $container['foobar']);
}
```

resolving 特性

resolving 回调的类型

resolving 不仅可以针对接口执行回调函数，还可以针对接口实现的类型进行回调函数。

```
public function testResolvingCallbacksAreCalledForType()
{
    $container = new Container;
    $container->resolving('StdClass', function ($object) {
        return $object->name = 'taylor';
    });
    $container->bind('foo', function () {
          return new StdClass;
    });
    $instance = $container->make('foo');

    $this->assertEquals('taylor', $instance->name);
}
public function testResolvingCallbacksShouldBeFiredWhenCalledWit
hAliases()
{
    $container = new Container;
    $container->alias('StdClass', 'std');
    $container->resolving('std', function ($object) {
        return $object->name = 'taylor';
    });
    $container->bind('foo', function () {
        return new StdClass;
    });
    $instance = $container->make('foo');

    $this->assertEquals('taylor', $instance->name);
}
```

resolving 回调与 instance

前面讲过，对于 singleton 绑定来说，resolving 回调函数仅仅运行一次，只在 singleton 第一次解析的时候才会调用。如果我们利用 instance 直接绑定类的对象，不需要解析，那么 resolving 回调函数将不会被调用：

```
public function testResolvingCallbacksAreCalledForSpecificAbstra
cts()
{
    $container = new Container;
    $container->resolving('foo', function ($object) {
        return $object->name = 'taylor';
    });
    $obj = new StdClass;
    $container->instance('foo', $obj);
    $instance = $container->make('foo');

    $this->assertFalse(isset($instance->name));
}
```

extend 扩展特性

extend 用于扩展绑定对象的功能，对于普通绑定来说，这个函数的位置很灵活：

在绑定前扩展

```
public function testExtendIsLazyInitialized()
{
    ContainerLazyExtendStub::$initialized = false;

    $container = new Container;
    $container->extend('Illuminate\Tests\Container\ContainerLazy
ExtendStub', function ($obj, $container) {
        $obj->init();
        return $obj;
    });
    $container->bind('Illuminate\Tests\Container\ContainerLazyEx
tendStub');

    $this->assertFalse(ContainerLazyExtendStub::$initialized);

    $container->make('Illuminate\Tests\Container\ContainerLazyEx
tendStub');
    $this->assertTrue(ContainerLazyExtendStub::$initialized);
}
```

在绑定后解析前扩展

```
public function testExtendIsLazyInitialized()
{
    ContainerLazyExtendStub::$initialized = false;

    $container = new Container;
    $container->bind('Illuminate\Tests\Container\ContainerLazyExtendStub');
    $container->extend('Illuminate\Tests\Container\ContainerLazyExtendStub', function ($obj, $container) {
        $obj->init();
        return $obj;
    });

    $this->assertFalse(ContainerLazyExtendStub::$initialized);

    $container->make('Illuminate\Tests\Container\ContainerLazyExtendStub');
    $this->assertTrue(ContainerLazyExtendStub::$initialized);
}
```

在解析后扩展

```
public function testExtendIsLazyInitialized()
{
    ContainerLazyExtendStub::$initialized = false;

    $container = new Container;
    $container->bind('Illuminate\Tests\Container\ContainerLazyExtendStub');

    $container->make('Illuminate\Tests\Container\ContainerLazyExtendStub');
    $this->assertFalse(ContainerLazyExtendStub::$initialized);

    $container->extend('Illuminate\Tests\Container\ContainerLazyExtendStub', function ($obj, $container) {
        $obj->init();
        return $obj;
    });
    $this->assertFalse(ContainerLazyExtendStub::$initialized);

    $container->make('Illuminate\Tests\Container\ContainerLazyExtendStub');
    $this->assertTrue(ContainerLazyExtendStub::$initialized);
}
```

可以看出，无论在哪个位置，extend 扩展都有 lazy 初始化的特点，也就是使用 extend 函数并不会立即起作用，而是要等到 make 解析才会激活。

extend 与 instance 绑定

对于 instance 绑定来说，暂时 extend 的位置需要位于 instance 之后才会起作用，并且会立即起作用，没有 lazy 的特点：

```
public function testExtendInstancesArePreserved()
{
    $container = new Container;

    $obj = new StdClass;
    $obj->foo = 'foo';
    $container->instance('foo', $obj);
    $container->extend('foo', function ($obj, $container) {
        $obj->bar = 'baz';

        return $obj;
    });

    $this->assertEquals('foo', $container->make('foo')->foo);
    $this->assertEquals('baz', $container->make('foo')->bar);
}
```

extend 绑定与 rebind 回调

无论扩展对象是 instance 绑定还是 bind 绑定，extend 都会启动 rebind 回调函数：

```
public function testExtendReBindingInstance()
{
    $_SERVER['_test_rebind'] = false;

    $container = new Container;
    $container->rebinding('foo',function (){
        $_SERVER['_test_rebind'] = true;
    });

    $obj = new StdClass;
    $container->instance('foo',$obj);

    $container->make('foo');

    $container->extend('foo', function ($obj, $container) {
        return $obj;
    });
```

```
    this->assertTrue($_SERVER['_test_rebind']);
}

public function testExtendReBinding()
{
    $_SERVER['_test_rebind'] = false;

    $container = new Container;
    $container->rebinding('foo',function (){
        $_SERVER['_test_rebind'] = true;
    });
    $container->bind('foo',function (){
        $obj = new StdClass;

        return $obj;
    });

    $container->make('foo');

    $container->extend('foo', function ($obj, $container) {
        return $obj;
    });

    this->assertFalse($_SERVER['_test_rebind']);
}
```

contextual 绑定特性

contextual 在绑定前

contextual 绑定不仅可以与 bind 绑定合作，相互不干扰，还可以与 instance 绑定相互合作。而且 instance 的位置也很灵活，可以在 contextual 绑定前，也可以在 contextual 绑定后：

```
public function testContextualBindingWorksForExistingInstancedBi
ndings()
{
    $container = new Container;

    $container->instance('Illuminate\Tests\Container\IContainerC
ontractStub', new ContainerImplementationStub);

    $container->when('Illuminate\Tests\Container\ContainerTestCo
ntextInjectOne')->needs('Illuminate\Tests\Container\IContainerCo
ntractStub')->give('Illuminate\Tests\Container\ContainerImplemen
tationStubTwo');

    $this->assertInstanceOf(
             'Illuminate\Tests\Container\ContainerImplementation
StubTwo',
             $container->make('Illuminate\Tests\Container\Contai
nerTestContextInjectOne')->impl
     );
}
```

contextual 在绑定后

```
public function testContextualBindingWorksForNewlyInstancedBindi
ngs()
{
    $container = new Container;

    $container->when('Illuminate\Tests\Container\ContainerTestCo
ntextInjectOne')->needs('Illuminate\Tests\Container\IContainerCo
ntractStub')->give('Illuminate\Tests\Container\ContainerImplemen
tationStubTwo');

    $container->instance('Illuminate\Tests\Container\IContainerC
ontractStub', new ContainerImplementationStub);

    $this->assertInstanceOf(
            'Illuminate\Tests\Container\ContainerImplementationS
tubTwo',
        $container->make('Illuminate\Tests\Container\ContainerTe
stContextInjectOne')->impl
    );
}
```

contextual 绑定与别名

contextual 绑定也可以在别名上进行，无论赋予别名的位置是 contextual 的前面还是后面：

```
public function testContextualBindingDoesntOverrideNonContextual
Resolution()
{
    $container = new Container;

    $container->instance('stub', new ContainerImplementationStub
);
    $container->alias('stub', 'Illuminate\Tests\Container\IConta
inerContractStub');

    $container->when('Illuminate\Tests\Container\ContainerTestCo
ntextInjectTwo')->needs('Illuminate\Tests\Container\IContainerCo
```

```
ntractStub')->give('Illuminate\Tests\Container\ContainerImplemen
tationStubTwo');

    $this->assertInstanceOf(
            'Illuminate\Tests\Container\ContainerImplementationS
tubTwo',
            $container->make('Illuminate\Tests\Container\Contain
erTestContextInjectTwo')->impl
        );

    $this->assertInstanceOf(
            'Illuminate\Tests\Container\ContainerImplementationS
tub',
            $container->make('Illuminate\Tests\Container\Contain
erTestContextInjectOne')->impl
    );
}

public function testContextualBindingWorksOnNewAliasedBindings()
{
    $container = new Container;

    $container->when('Illuminate\Tests\Container\ContainerTestCo
ntextInjectOne')->needs('Illuminate\Tests\Container\IContainerCo
ntractStub')->give('Illuminate\Tests\Container\ContainerImplemen
tationStubTwo');

    $container->bind('stub', ContainerImplementationStub::class)
;
    $container->alias('stub', 'Illuminate\Tests\Container\IConta
inerContractStub');

    $this->assertInstanceOf(
          'Illuminate\Tests\Container\ContainerImplementationStu
bTwo',
          $container->make('Illuminate\Tests\Container\Container
TestContextInjectOne')->impl
    );
}
```

争议

目前比较有争议的是下面的情况：

```
public function testContextualBindingWorksOnExistingAliasedInsta
nces()
{
    $container = new Container;

    $container->alias('Illuminate\Tests\Container\IContainerCont
ractStub', 'stub');
    $container->instance('stub', new ContainerImplementationStub
);

    $container->when('Illuminate\Tests\Container\ContainerTestCo
ntextInjectOne')->needs('stub')->give('Illuminate\Tests\Containe
r\ContainerImplementationStubTwo');

    $this->assertInstanceOf(
        'Illuminate\Tests\Container\ContainerImplementationStubT
wo',
        $container->make('Illuminate\Tests\Container\ContainerTe
stContextInjectOne')->impl
    );
}
```

由于instance的特性，当别名被绑定到其他对象上时，别名 stub 已经失去了与 Illuminate\Tests\Container\IContainerContractStub 之间的关系，因此不能使用 stub 代替作上下文绑定。但是另一方面：

```
public function testContextualBindingWorksOnBoundAlias()
{
    $container = new Container;

    $container->alias('Illuminate\Tests\Container\IContainerCont
ractStub', 'stub');
    $container->bind('stub', ContainerImplementationStub::class)
;

    $container->when('Illuminate\Tests\Container\ContainerTestCo
ntextInjectOne')->needs('stub')->give('Illuminate\Tests\Containe
r\ContainerImplementationStubTwo');

    $this->assertInstanceOf(
        'Illuminate\Tests\Container\ContainerImplementationStubT
wo',
        $container->make('Illuminate\Tests\Container\ContainerTe
stContextInjectOne')->impl
    );
}
```

代码只是从 instance 绑定改为 bind 绑定，由于 bind 绑定只切断了别名中的 alias 数组的联系，并没有断绝abstractAlias数组的联系，因此这段代码却可以通过，很让人难以理解。本人在给 Taylor Otwell 提出 PR 时，作者原话为“I'm not making any of these changes to the container on a patch release.”。也许，在以后(5.5或以后)版本作者会更新这里的逻辑，我们就可以看看服务容器对别名绑定的态度了，大家也最好不要这样用。

服务容器中的闭包函数参数

服务容器中很多函数都有闭包函数，这些闭包函数可以放入特定的参数，在绑定或者解析过程中，这些参数会被服务容器自动带入各种类对象或者服务容器实例。

bind 闭包参数

```
public function testAliasesWithArrayOfParameters()
{
    $container = new Container;
    $container->bind('foo', function ($app, $config) {
        return $config;
    });

    $container->alias('foo', 'baz');
    $this->assertEquals([1, 2, 3], $container->makeWith('baz', [1
, 2, 3]));
}
```

extend 闭包参数

```
public function testExtendedBindings()
{
    $container = new Container;
    $container['foo'] = 'foo';
    $container->extend('foo', function ($old, $container) {
        return $old.'bar';
    });

    $this->assertEquals('foobar', $container->make('foo'));

    $container = new Container;

    $container->singleton('foo', function () {
        return (object) ['name' => 'taylor'];
    });
    $container->extend('foo', function ($old, $container) {
        $old->age = 26;
        return $old;
    });

    $result = $container->make('foo');
    $this->assertEquals('taylor', $result->name);
    $this->assertEquals(26, $result->age);
    $this->assertSame($result, $container->make('foo'));
}
```

bindmethod 闭包参数

```
public function testCallWithBoundMethod()
{
    $container = new Container;
    $container->bindMethod('Illuminate\Tests\Container\Container
TestCallStub@unresolvable', function ($stub,$container) {
        $container['foo'] = 'foo';
        return $stub->unresolvable('foo', 'bar');
    });
    $result = $container->call('Illuminate\Tests\Container\Conta
inerTestCallStub@unresolvable');
    $this->assertEquals(['foo', 'bar'], $result);
    $this->assertEquals('foo',$container['foo']);
}
```

resolve 闭包参数

```
public function testResolvingCallbacksAreCalledForSpecificAbstra
cts()
{
     $container = new Container;
     $container->resolving('foo', function ($object，$container)
{
         return $object->name = 'taylor';
     });

     $container->bind('foo', function () {
        return new StdClass;
     });
     $instance = $container->make('foo');

     $this->assertEquals('taylor', $instance->name);
}
```

rebinding 闭包参数

```php
public function testReboundListeners()
{
    $container = new Container;
    $container->bind('foo', function () {
        return 'foo';
    });

    $container->rebinding('foo', function ($container,$object) {
         $container['bar'] = $object.'bar';
    });

    $container->bind('foo', function () {
    });

    $this->assertEquals('bar',$container['foobar']);
}
```

前言

作为一个 web 后台框架，路由无疑是极其重要的一部分。本博客接下来几篇文章都将会围绕路由这一主题来展开讨论，分别讲述：

> - 路由的使用
> - 路由属性注册
> - 路由的正则编译与匹配
> - 路由的中间件
> - 路由的控制器与参数绑定
> - RESTful 路由

和之前一样，第一篇将会利用单元测试样例说明我们在平时可能用到的 route 的 api 函数用法，后面几篇文章将会剖析 laravel 的 route 源码。下面开始介绍 laravel 中路由的各种用法。

路由属性注册

所有 Laravel 路由都定义在位于 routes 目录下的路由文件中，这些文件通过框架自动加载。routes/web.php 文件定义了 web 界面的路由，这些路由被分配了 web 中间件组，从而可以提供 session 和 csrf 防护等功能。routes/api.php 中的路由是无状态的，被分配了 api 中间件组。

对大多数应用而言，都是从 routes/web.php 文件开始定义路由。

路由 method 方法

我们可以注册路由来响应任何 HTTP 请求：

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri, $callback);
```

有时候还需要注册路由响应多个 HTTP 请求——这可以通过 `match` 方法来实现。或者，可以使用 `any` 方法注册一个路由来响应所有 HTTP 请求：

```
Route::match(['get', 'post'], '/', function () {
    //
});

Route::any('foo', function () {
    //
});
```

值得注意的是，一般的HTML表单仅仅支持 `get` 、 `post` ，并不支持 `put` 、 `patch` 、 `delete` 等动作，这时候就需要在前端添加一个隐藏的 `_method` 字段到给表单中，其值被用作 `HTTP` 请求方法名：

```
<input type="hidden" name="_method" value="PUT">
```

在 web 路由文件中所有请求方式为 `PUT` 、 `POST` 或 `DELETE` 的 HTML 表单都会包含一个 `CSRF` 令牌字段，否则，请求会被拒绝。关于 `CSRF` 的更多细节，可以参考 浅谈CSRF攻击方式：

```
<form method="POST" action="/profile">
    {{ csrf_field() }}
    ...
</form>
```

路由 scheme 协议

对于 web 后台框架来说，路由的 `scheme` 底层协议一般使用 `http` 、 `https` ：

```
Route::get('foo/{bar}', ['http', function () {
    }]);
Route::get('foo/{bar}', ['https', function () {
    }]);
```

路由 domain 子域名

子域名可以像 URI 一样被分配给路由参数，子域名可以通过路由属性中的 `domain` 来指定：

```
Route::domain('api.name.bar')
     ->get('foo/bar', function ($name) {
         return $name;
    });

Route::get('foo/bar', ['domain' => 'api.name.bar', function ($na
me) {
         return $name;
    }]);
```

路由 prefix 前缀

可以为路由添加一个给定 `URI` 前缀，通过利用路由属性的 `prefix` 指定：

```
Route::prefix('pre')
     ->get('foo/bar', function () {
    });

Route::get('foo/bar', ['prefix' => 'pre', function () {
    }]);

Route::get('foo/bar', function () {
    })->prefix('pre');
```

路由 where 正则约束

可以为路由的 `URI` 参数指定正则约束：

```
Route::get('{one}', ['where' => ['one' => '(.+)'], function () {
    }]);

Route::get('{one}', function () {
    })->where('one', '(.+)');
```

如果想要路由参数在全局范围内被给定正则表达式约束，可以使用 `pattern` 方法。在 `RouteServiceProvider` 类的 `boot` 方法中定义约束模式：

```
public function boot()
{
    Route::pattern('one', '(.+)');
    parent::boot();
}
```

路由 middleware 中间件

为路由添加中间件，通过利用路由属性的 `middleware` 指定：

```
Route::middleware('web')
     ->get('foo/bar', function () {
    });

Route::get('foo/bar', ['middleware' => 'web', function () {
    }]);

Route::get('foo/bar', function () {
    })->middleware('web');
```

路由 namespace 属性

可以为路由的控制器添加 `namespace` 来指定控制器的命名空间：

```
Route::namespace('Namespace\\Example\\')
     ->get('foo/bar', function () {
    });

Route::get('foo/bar', ['namespace' => 'Namespace\\Example\\', fu
nction () {
    }]);
```

路由 uses 属性

可以为路由添加 `URI` 对应的执行逻辑，例如闭包或者控制器：

```
Route::get('foo/bar', ['uses' => function () {
    }]);

Route::get('foo/bar', ['uses' => ‘Illuminate\Tests\Routing\Route
TestControllerStub@index’]);

Route::get('foo/bar')->uses(function () {
    });

Route::get('foo/bar')->uses(‘Illuminate\Tests\Routing\RouteTestC
ontrollerStub@index’);
```

路由 as 别名

可以为路由指定别名，通过路由属性的 `as` 来指定：

```
Route::as('Foo')
     ->get('foo/bar', function () {
    });

Route::name('Foo')
     ->get('foo/bar', function () {
    });

Route::get('foo/bar', ['as' => 'Foo', function () {
    }]);

Route::get('foo/bar', function () {
    })->name('Foo');
```

路由 group 群组属性

可以为一系列具有类似属性的路由归为同一组，利用 `group` 将这些路由归并到一起：

```
Route::group(['domain'     => 'group.domain.name',
              'prefix'     => 'grouppre',
              'where'      => ['one' => '(.+)'],
              'middleware' => 'groupMiddleware',
              'namespace'  => 'Namespace\\Group\\',
              'as'         => 'Group::',]
              function () {
                  Route::get('/replace',‘domain’ => 'route.domai
n.name'，
                                        'uses'    => function ()
{
                                        return 'replace';
                  });

                 Route::get('additional/{one}/{two}', 'prefix'
   => 'routepre',
                                                       'where'
   => '['one' => '([0-9]+)','two' => '(.+)']',
                                                       'middlewar
```

```
e' => 'routeMiddleware',
                                                      'namespace'
  => 'Namespace\\Group\\',
                                                      'as'
   => 'Route',
                                                      'use
   => 'function () {
                                                      return 'ad
ditional';
                 });
});

$this->assertEquals('replace', $router->dispatch(Request::create(
'http://route.domain.name/grouppre/replace', 'GET'))->getContent
());

$this->assertEquals('additional', $router->dispatch(Request::cre
ate('http://group.domain.name/routepre/grouppre/additional/111/a
dd', 'GET'))->getContent());

$routes = $router->getRoutes()->getRoutes();
$action = $routes[0]->getAction();
$this->assertEquals('Namespace\\Group\\', $action['namespace']);
$this->assertEquals('Group::', $action['as']);

$routes = $router->getRoutes()->getRoutes();
$action = $routes[1]->getAction();
$this->assertEquals(['groupMiddleware', 'routeMiddleware'], $act
ion['middleware']);
$this->assertEquals('Namespace\\Group\\Namespace\\Group\\', $act
ion['namespace']);
$this->assertEquals('Group::Route', $action['as']);
```

`group` 群组的属性分为两类：替换型、递增型。当群组属性与路由属性重复的时候，替换型属性会用路由的属性替换群组的属性，递增型的属性会综合路由和群组的属性。

在上面的例子可以看出：

- `domain` 这个属性是替换型属性，路由的属性会覆盖和替换群组的这几个属性；
- `prefix` 、 `middleware` 、 `namespace` 、 `as` 、 `where` 这几个属性是递增型属性，路由的属性和群组属性会相互结合。

另外值得注意的是：

- 路由的 `prefix` 属性具有优先级,因此上面第二个路由的 `uri` 是 `routepre/grouppre/additional/111/add` ,而不是 `grouppre/routepre/additional/111/add` ；
- `where` 属性对于相同的路由参数会替换，不同的路由参数会结合，因此上面 `where` 中 `one` 被替换， `two` 被结合进来

路由参数与匹配

laravel 允许在注册定义路由的时候设定路由参数，以供控制器或者闭包所用。路由参数可以设定在 `URI` 中，也可以设定在 `domain` 中。

路由编码匹配

对于已编码的请求 `URI` ，框架会自动进行解码然后进行匹配:

```
$router = $this->getRouter();
 $router->get('foo/bar/åαф', function () {
     return 'hello';
});
$this->assertEquals('hello', $router->dispatch(Request::create('
foo/bar/%C3%A5%CE%B1%D1%84', 'GET'))->getContent());

$router = $this->getRouter();
$route = $router->get('foo/{file}', function ($file) {
    return $file;
});
$this->assertEquals('oxygen%20', $router->dispatch(Request::crea
te('http://test.com/foo/oxygen%2520', 'GET'))->getContent());
```

路由参数

路由参数总是通过花括号进行包裹，这些参数在路由被执行时会被传递到路由的闭包。路由参数不能包含 - 字符，需要的话可以使用 _ 替代。

```
$router = $this->getRouter();
$route = $router->get('foo/{age}', ['domain' => 'api.{name}.bar'
, function ($name, $age) {
   return $name.$age;
}]);
$this->assertEquals('taylor25', $router->dispatch(Request::creat
e('http://api.taylor.bar/foo/25', 'GET'))->getContent());

$route = new Route('GET', 'images/{id}.{ext}', function () {
    });

$request1 = Request::create('images/1.png', 'GET');
$this->assertTrue($route->matches($request1));
$route->bind($request1);
$this->assertTrue($route->hasParameter('id'));
$this->assertFalse($route->hasParameter('foo'));
$this->assertEquals('1', $route->parameter('id'));
$this->assertEquals('png', $route->parameter('ext'));
```

路由可选参数

有时候可能需要指定可选的路由参数，这可以通过在参数名后加一个 ? 标记来实现，这种情况下需要给相应的变量指定默认值：

```
$router = $this->getRouter();
$router->get('{foo?}/{baz?}', function ($name = 'taylor', $age =
25) {
     return $name.$age;
});
$this->assertEquals('fred25', $router->dispatch(Request::create(
'fred', 'GET'))->getContent());

$router->get('default/{foo?}/{baz?}', function ($name, $age = 25)
 {
     return $name.$age;
})->default('name', 'taylor');
$this->assertEquals('fred25', $router->dispatch(Request::create(
'fred', 'GET'))->getContent());
```

路由参数正则约束

可以使用路由实例上的 `where` 方法来约束路由参数的格式。 `where` 方法接收参数名和一个正则表达式来定义该参数如何被约束：

```
Route::get('user/{name}', function ($name) {
    //
})->where('name', '[A-Za-z]+');
```

如果想要路由参数在全局范围内被给定正则表达式约束，可以使用 `pattern` 方法。在 `RouteServiceProvider` 类的 `boot` 方法中定义约束模式:

```
public function boot()
{
    Route::pattern('id', '[0-9]+');
    parent::boot();
}
```

值得注意的是，路由参数是不允许出现 `/` 字符的，例如：

```
$router->get('{one?}', [
            'uses' => function ($one = null){
                return $one;
            },
        ]);
$request = Request::create('foo/bar/baz', 'GET');
$this->assertFalse($route->matches($request2));
```

上例中 `one` 只能匹配 `foo` ，不能匹配 `foo/bar/baz` ,这时就需要对 `one` 进行正则约束：

```
public function testLeadingParamDoesntReceiveForwardSlashOnEmpty
Path()
{
    $router = $this->getRouter();
    $router->get('{one?}', [
            'uses' => function ($one = null){
                return $one;
            },
            'where' => ['one' => '(.+)'],
        ]);

    $this->assertEquals('foo', $router->dispatch(Request::create(
'/foo', 'GET'))->getContent());
    $this->assertEquals('foo/bar/baz', $router->dispatch(Request
::create('/foo/bar/baz', 'GET'))->getContent());
}
```

路由中间件

HTTP 中间件为过滤进入应用的 HTTP 请求提供了一套便利的机制。例如，Laravel 内置了一个中间件来验证用户是否经过认证，如果用户没有经过认证，中间件会将用户重定向到登录页面，否则如果用户经过认证，中间件就会允许请求继续往前进入下一步操作。

Laravel框架自带了一些中间件，包括认证、CSRF 保护中间件等等。所有的中间件都位于 app/Http/Middleware 目录。

中间件之前/之后/终止

一个中间件是请求前还是请求后执行取决于中间件本身。比如，以下中间件会在请求处理前执行一些任务：

```
class BeforeMiddleware
{
    public function handle($request, Closure $next)
    {
        // 执行动作

        return $next($request);
    }
}

class AfterMiddleware
{
    public function handle($request, Closure $next)
    {
        $response = $next($request);

        // 执行动作

        return $response;
    }
}
```

有时候中间件可能需要在 HTTP 响应发送到浏览器之后做一些工作。比如，Laravel 内置的“session”中间件会在响应发送到浏览器之后将 Session 数据写到存储器中，为了实现这个功能，需要定义一个终止中间件并添加 terminate 方法到这个中间件：

```
class StartSession
{
    public function handle($request, Closure $next)
    {
        return $next($request);
    }

    public function terminate($request, $response)
    {
        // 存储session数据...
    }
}
```

全局中间件

如果你想要中间件在每一个 HTTP 请求期间被执行，只需要将相应的中间件类设置到 `app/Http/Kernel.php` 的数组属性 `$middleware` 中即可。

```
protected $middleware = [
        \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
        \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
        \App\Http\Middleware\TrimStrings::class,
        \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
    ];
```

路由中间件

如果你想要分配中间件到指定路由，可以传递完整的类名：

```
use App\Http\Middleware\CheckAge;

Route::get('admin/profile', function () {
    //
})->middleware(CheckAge::class);
```

或者可以给中间件提供一个别名：

```
public function testDefinedClosureMiddleware()
 {
    $router = $this->getRouter();
    $router->get('foo/bar', ['middleware' => 'foo', function ()
{
        return 'hello';
    }]);
    $router->aliasMiddleware('foo', function ($request, $next) {
        return 'caught';
    });
    $this->assertEquals('caught', $router->dispatch(Request::cre
ate('foo/bar', 'GET'))->getContent());
 }
```

也可以应该在 `app/Http/Kernel.php` 文件中分配给该中间件一个 `key` ，默认情况下，该类的 `$routeMiddleware` 属性包含了 `Laravel` 自带的中间件，要添加你自己的中间件，只需要将其追加到后面并为其分配一个 `key` ，例如：

```
protected $routeMiddleware = [
    'auth' => \Illuminate\Auth\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWith
BasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindi
ngs::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::cla
ss,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequest
s::class,
];

Route::get('admin/profile', function () {
    //
})->middleware('auth');
```

使用数组分配多个中间件到路由：

```
Route::get('/', function () {
    //
})->middleware('first', 'second');
```

中间件组

有时候你可能想要通过指定一个键名的方式将相关中间件分到同一个组里面，从而更方便将其分配到路由中，这可以通过使用 `HTTP Kernel` 的 `$middlewareGroups` 属性实现。

`Laravel` 自带了开箱即用的 `web` 和 `api` 两个中间件组以分别包含可以应用到 `Web UI` 和 `API` 路由的通用中间件：

```
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse
::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::clas
s,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class
,
    ],

    'api' => [
        'throttle:60,1',
        'auth:api',
    ],
];

Route::get('/', function () {
    //
})->middleware('web');
```

值得注意的是，中间件组中可以循环嵌套中间件组：

```
 public function testMiddlewareGroupsCanReferenceOtherGroups()
 {
    unset($_SERVER['__middleware.group']);
    $router = $this->getRouter();
    $router->get('foo/bar', ['middleware' => 'web', function ()
{
        return 'hello';
    }]);

    $router->aliasMiddleware('two', 'Illuminate\Tests\Routing\Ro
utingTestMiddlewareGroupTwo');
    $router->middlewareGroup('first', ['two:abigail']);
    $router->middlewareGroup('web', ['Illuminate\Tests\Routing\R
outingTestMiddlewareGroupOne', 'first']);

    $this->assertEquals('caught abigail', $router->dispatch(Requ
est::create('foo/bar', 'GET'))->getContent());
    $this->assertTrue($_SERVER['__middleware.group']);

    unset($_SERVER['__middleware.group']);
}
```

中间件参数

中间件还可以接收额外的自定义参数，例如，如果应用需要在执行给定动作之前验证认证用户是否拥有指定的角色，可以创建一个 `CheckRole` 来接收角色名作为额外参数。

额外的中间件参数会在 `$next` 参数之后传入中间件：

```
namespace App\Http\Middleware;

use Closure;

class CheckRole
{
    public function handle($request, Closure $next, $role)
    {
        if (! $request->user()->hasRole($role)) {
            // Redirect...
        }

        return $next($request);
    }

}

Route::put('post/{id}', function ($id) {
    //
})->middleware('role:editor');
```

中间件的顺序

当 `router` 中有多个中间件的时候，中间件的执行顺序并不是严格按照中间件数组进行的，框架中存在一个数组 `$middlewarePriority` ,规定了这个数组中各个中间件的顺序：

```
protected $middlewarePriority = [
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::clas
s,
        \Illuminate\Auth\Middleware\Authenticate::class,
        \Illuminate\Session\Middleware\AuthenticateSession::clas
s,
        \Illuminate\Routing\Middleware\SubstituteBindings::class
,
        \Illuminate\Auth\Middleware\Authorize::class,
    ];
```

当我们使用了上面其中多个中间件的时候，框架会自动按照上面的数组进行排序：

```
 public function testMiddlewarePrioritySorting()
{
    $middleware = [
        Placeholder1::class,
        SubstituteBindings::class,
        Placeholder2::class,
        Authenticate::class,
        Placeholder3::class,
    ];

    $router = $this->getRouter();

    $router->middlewarePriority = [Authenticate::class, Substitu
teBindings::class, Authorize::class];

    $route = $router->get('foo', ['middleware' => $middleware, '
uses' => function ($name) {
        return $name;
    }]);

    $this->assertEquals([
        Placeholder1::class,
        Authenticate::class,
        SubstituteBindings::class,
        Placeholder2::class,
        Placeholder3::class,
    ], $router->gatherRouteMiddleware($route));
}
```

控制器

控制器类

更普遍的方法是使用控制器来组织管理这些行为。控制器可以将相关的 `HTTP` 请求封装到一个类中进行处理。通常控制器存放在 `app/Http/Controllers` 目录中.

所有的 Laravel 控制器应该继承自 Laravel 自带的控制器基类 `Controller`，控制器基类提供了一些很方便的方法如 `middleware`，用于添加中间件到控制器动作：

```
class UserController extends Controller
{
    public function show($id)
    {
        return view('user.profile', ['user' => User::findOrFail(
$id)]);
    }
}

Route::get('user/{id}', 'UserController@show');
```

单动作控制器

如果想要定义一个只处理一个动作的控制器，可以在这个控制器中定义 __invoke 方法,当为这个单动作控制器注册路由的时候，不需要指定方法：

```
public function testDispatchingCallableActionClasses()
{
    $router = $this->getRouter();
    $router->get('foo/bar', 'Illuminate\Tests\Routing\ActionStub'
);

    $this->assertEquals('hello', $router->dispatch(Request::crea
te('foo/bar', 'GET'))->getContent());

    $router->get('foo/bar2', [
        'uses' => 'Illuminate\Tests\Routing\ActionStub@func',
    ]);

    $this->assertEquals('hello2', $router->dispatch(Request::cre
ate('foo/bar2', 'GET'))->getContent());
}

class ActionStub extends Controller
{
    public function __invoke()
    {
        return 'hello';
    }
}
```

控制器中间件

将中间件放在控制器构造函数中更方便，在控制器的构造函数中使用 middleware 方法你可以很轻松的分配中间件给该控制器。你甚至可以限定该中间件应用到该控制器类的指定方法：

```
class UserController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth');
        $this->middleware('log')->only('index');
        $this->middleware('subscribed')->except('store');
    }
}
```

callAction 方法

值得注意的是每次执行控制器方法都会先执行控制器的 `callAction` 函数：

```
public function callAction($method, $parameters)
{
    return call_user_func_array([$this, $method], $parameters);
}
```

测试样例：

```
unset($_SERVER['__test.controller_callAction_parameters']);
$router->get(($str = str_random()).'/{one}/{two}', 'Illuminate\T
ests\Routing\RouteTestAnotherControllerWithParameterStub@oneArgu
ment');
$router->dispatch(Request::create($str.'/one/two', 'GET'));
 $this->assertEquals(['one' => 'one', 'two' => 'two'], $_SERVER[
'__test.controller_callAction_parameters']);


class RouteTestAnotherControllerWithParameterStub extends Contro
ller
{
    public function callAction($method, $parameters)
    {
        $_SERVER['__test.controller_callAction_parameters'] = $p
arameters;
    }

    public function oneArgument($one)
    {
    }
}
```

__call方法

和普通类一样，若控制器中没有对应 `classname@method` 中的 `method` ,则会调用类的 `__call` 函数。

```
public function testCallableControllerRouting()
{
    $router = $this->getRouter();

    $router->get('foo/bar', 'Illuminate\Tests\Routing\RouteTestC
ontrollerCallableStub@bar');
    $router->get('foo/baz', 'Illuminate\Tests\Routing\RouteTestC
ontrollerCallableStub@baz');

    $this->assertEquals('bar', $router->dispatch(Request::create(
'foo/bar', 'GET'))->getContent());
    $this->assertEquals('baz', $router->dispatch(Request::create(
'foo/baz', 'GET'))->getContent());
}

class RouteTestControllerCallableStub extends Controller
{
    public function __call($method, $arguments = [])
    {
        return $method;
    }
}
```

路由参数依赖注入与绑定

Laravel 使用服务容器解析所有的 Laravel 控制器，因此，可以在控制器的构造函数中类型声明任何依赖，这些依赖会被自动解析并注入到控制器实例中。路由的参数绑定可以分为两种：显示绑定与隐示绑定。

路由隐示绑定

- 控制器方法期望输入路由参数，只需要将路由参数放到其他依赖之后

```
Route::put('user/{id}', 'UserController@update');

class UserController extends Controller
{
    public function update(Request $request, $id)
    {
    }
}
```

- 可以在控制器的动作方法中进行依赖的类型提示，例如，我们可以在某个方法中类型提示 `Illuminate\Http\Request` 实例：

```
class UserController extends Controller
{
    public function store(Request $request)
    {
        $name = $request->input('name');
    }
}
```

- 可以为控制器的动作方法中添加数据库模型的主键，框架会自动利用主键来获取对应的记录，需要注意的是，route定义路由的路由参数必须和控制器内的变量名相同，例如下例中路由参数 `userid` 和控制器参数 `userid`：

```
Route::put('user/{userid}', 'UserController@update');

class UserController extends Controller
{
    public function update(UserModel $userid)
    {
        $userid->name = 'taylor';
        $userid->update();
    }
}
```

综合测试样例：

```
public function testImplicitBindingsWithOptionalParameter()
{
    unset($_SERVER['__test.controller_callAction_parameters']);
    $router->get(($str = str_random()).'/{user}/{defaultNull?}/{
team?}', [
        'middleware' => SubstituteBindings::class,
        'uses' => 'Illuminate\Tests\Routing\RouteTestAnotherCont
rollerWithParameterStub@withModels',
    ]);

    $router->dispatch(Request::create($str.'/1', 'GET'));

    $values = array_values($_SERVER['__test.controller_callActio
n_parameters']);

    $this->assertInstanceOf('Illuminate\Http\Request', $values[0
]);
    $this->assertEquals(1, $values[1]->value);
    $this->assertNull($values[2]);
    $this->assertInstanceOf('Illuminate\Tests\Routing\RoutingTes
tTeamModel', $values[3]);
}


class RouteTestAnotherControllerWithParameterStub extends Contro
ller
{
    public function callAction($method, $parameters)
    {
        $_SERVER['__test.controller_callAction_parameters'] = $p
arameters;
    }

    public function withModels(Request $request, RoutingTestUser
Model $user, $defaultNull = null, RoutingTestTeamModel $team = n
ull)
    {
    }
}
```

```
class RoutingTestUserModel extends Model
{
    public function getRouteKeyName()
    {
        return 'id';
    }

    public function where($key, $value)
    {
        $this->value = $value;

        return $this;
    }

    public function first()
    {
        return $this;
    }

    public function firstOrFail()
    {
        return $this;
    }
}

class RoutingTestTeamModel extends Model
{
    public function getRouteKeyName()
    {
        return 'id';
    }

    public function where($key, $value)
    {
        $this->value = $value;

        return $this;
    }
```

```
    public function first()
    {
        return $this;
    }

    public function firstOrFail()
    {
        return $this;
    }
}
```

路由显示绑定

除了隐示地转化路由参数外，我们还可以给路由参数显示提供绑定。显示绑定有 `bind` 、 `model` 两种方法。

- 通过 `bind` 为参数绑定闭包函数：

```
public function testRouteBinding()
{
    $router = $this->getRouter();
    $router->get('foo/{bar}', ['middleware' => SubstituteBinding
s::class, 'uses' => function ($name) {
         return $name;
    }]);
    $router->bind('bar', function ($value) {
        return strtoupper($value);
    });
    $this->assertEquals('TAYLOR', $router->dispatch(Request::cre
ate('foo/taylor', 'GET'))->getContent());
}
```

- 通过 `bind` 为参数绑定类方法，可以指定 `classname@method` ，也可以直接使用类名，默认会调用类的 `bind` 函数：

```
public function testRouteClassBinding()
{
    $router = $this->getRouter();
    $router->get('foo/{bar}', ['middleware' => SubstituteBindings::class, 'uses' => function ($name) {
        return $name;
    }]);
    $router->bind('bar', 'Illuminate\Tests\Routing\RouteBindingStub');
    $this->assertEquals('TAYLOR', $router->dispatch(Request::create('foo/taylor', 'GET'))->getContent());
}

public function testRouteClassMethodBinding()
{
    $router = $this->getRouter();
    $router->get('foo/{bar}', ['middleware' => SubstituteBindings::class, 'uses' => function ($name) {
        return $name;
    }]);
    $router->bind('bar', 'Illuminate\Tests\Routing\RouteBindingStub@find');
    $this->assertEquals('dragon', $router->dispatch(Request::create('foo/Dragon', 'GET'))->getContent());
}

class RouteBindingStub
{
    public function bind($value, $route)
    {
        return strtoupper($value);
    }

    public function find($value, $route)
    {
        return strtolower($value);
    }
}
```

- 通过 `model` 为参数绑定数据库模型，路由的参数就不需要和控制器方法中的变量名相同，`laravel` 会利用路由参数的值去调用 `where` 方法查找对应记录：

```
if ($model = $instance->where($instance->getRouteKeyName(), $value)->first()) {
     return $model;
}
```

测试样例如下：

```
public function testModelBinding()
{
    $router = $this->getRouter();
    $router->get('foo/{bar}', ['middleware' => SubstituteBinding
s::class, 'uses' => function ($name) {
        return $name;
    }]);
    $router->model('bar', 'Illuminate\Tests\Routing\RouteModelBi
ndingStub');
    $this->assertEquals('TAYLOR', $router->dispatch(Request::cre
ate('foo/taylor', 'GET'))->getContent());
}

class RouteModelBindingStub
{
    public function getRouteKeyName()
    {
        return 'id';
    }

    public function where($key, $value)
    {
        $this->value = $value;

        return $this;
    }

    public function first()
    {
        return strtoupper($this->value);
    }
}
```

- 若绑定的 `model` 并没有找到对应路由参数的记录，可以在 `model` 中定义一个闭包函数，路由参数会调用闭包函数：

```
public function testModelBindingWithCustomNullReturn()
{
    $router = $this->getRouter();
```

```
    $router->get('foo/{bar}', ['middleware' => SubstituteBindings::class, 'uses' => function ($name) {
        return $name;
    }]);
    $router->model('bar', 'Illuminate\Tests\Routing\RouteModelBindingNullStub', function () {
        return 'missing';
    });
    $this->assertEquals('missing', $router->dispatch(Request::create('foo/taylor', 'GET'))->getContent());
}

public function testModelBindingWithBindingClosure()
{
    $router = $this->getRouter();
    $router->get('foo/{bar}', ['middleware' => SubstituteBindings::class, 'uses' => function ($name) {
        return $name;
    }]);
    $router->model('bar', 'Illuminate\Tests\Routing\RouteModelBindingNullStub', function ($value) {
        return (new RouteModelBindingClosureStub())->findAlternate($value);
    });
    $this->assertEquals('tayloralt', $router->dispatch(Request::create('foo/TAYLOR', 'GET'))->getContent());
}

class RouteModelBindingNullStub
{
    public function getRouteKeyName()
    {
        return 'id';
    }

    public function where($key, $value)
    {
        return $this;
    }
```

```
    public function first()
    {
    }
}

class RouteModelBindingClosureStub
{
    public function findAlternate($value)
    {
        return strtolower($value).'alt';
    }
}
```

router扩展方法

router支持添加自定义的方法，只需要利用 `macro` 函数来注册对应的函数名和函数实现：

```
public function testMacro()
{
    $router = $this->getRouter();
    $router->macro('webhook', function () use ($router) {
        $router->match(['GET', 'POST'], 'webhook', function () {
            return 'OK';
        });
    });
    $router->webhook();
    $this->assertEquals('OK', $router->dispatch(Request::create(
'webhook', 'GET'))->getContent());
    $this->assertEquals('OK', $router->dispatch(Request::create(
'webhook', 'POST'))->getContent());
}
```

前言

作为 `laravel` 极其重要的一部分， `route` 功能贯穿着整个网络请求，是 `request` 生命周期的主干。本文主要讲述 `route` 服务的注册与启动、路由的属性注册。本篇内容相对简单，更多的是框架添加路由的整体设计流程。

route 服务的注册

`laravel` 在接受到请求后，先进行了服务容器与 `http` 核心的初始化，再进行了请求 `request` 的构造与分发。

`route` 服务的注册—— `RoutingServiceProvider` 发生在服务容器 `container` 的初始化上;

`route` 服务的启动与加载—— `RouteServiceProvider` 发生在 `request` 的分发上。

route 服务的注册——RoutingServiceProvider

所有需要 `laravel` 服务的请求都会加载入口文件 `index.php` :

```
require __DIR__.'/../bootstrap/autoload.php';

$app = require_once __DIR__.'/../bootstrap/app.php';
```

第一句我们在之前的博客提过，是实现 `PSR0` 、 `PSR4` 标准自动加载的功能模块，第二句就是今天说的 `container` 的初始化：

```
$app = new Illuminate\Foundation\Application(
    realpath(__DIR__.'/../')
);
```

`Application` ：

```
namespace Illuminate\Foundation;

class Application extends Container implements ApplicationContra
ct, HttpKernelInterface
{
    public function __construct($basePath = null)
    {
        if ($basePath) {
            $this->setBasePath($basePath);
        }

        $this->registerBaseBindings();

        $this->registerBaseServiceProviders();

        $this->registerCoreContainerAliases();
    }
}
```

路由服务的注册就在 `registerBaseServiceProviders()` 这个函数中：

```
protected function registerBaseServiceProviders()
{
    $this->register(new EventServiceProvider($this));

    $this->register(new LogServiceProvider($this));

    $this->register(new RoutingServiceProvider($this));
}
```

`RoutingServiceProvider` :

```
namespace Illuminate\Routing;

class RoutingServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->registerRouter();

        ...
    }

    protected function registerRouter()
    {
        $this->app->singleton('router', function ($app) {
            return new Router($app['events'], $app);
        });
    }

    ...

}
```

可以看到， `RoutingServiceProvider` 做的事情比较简单，就是向服务容易中注册 `router` 。

`route` 服务的启动与加载——RouteServiceProvider

`laravel` 在初始化 `Application` 后，就要进行 `http/Kernel` 的构造：

```
$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);

$response = $kernel->handle(
    $request = Illuminate\Http\Request::capture()
);
```

初始化结束后，就会调用 `handle` 函数，这个函数用于 `laravel` 各个功能服务的注册启动，还有 `request` 的分发：

```
public function handle($request)
{
    try {
        $request->enableHttpMethodParameterOverride();

        $response = $this->sendRequestThroughRouter($request);
    }

    return $response;
}

protected function sendRequestThroughRouter($request)
{
    $this->app->instance('request', $request);

    Facade::clearResolvedInstance('request');

    $this->bootstrap();//各种服务的注册与启动

    return (new Pipeline($this->app))//请求的分发
                ->send($request)
                ->through($this->app->shouldSkipMiddleware() ? [
] : $this->middleware)
                ->then($this->dispatchToRouter());
    }
```

路由服务的启动与加载就在其中一个函数中 `bootstrap`，这个函数用于各种服务的注册与启动，比较复杂，我们有机会在以后单独来说。

总之，这个函数会调用 `RouteServiceProvider` 这个类的两个函数：注册——`register`、启动——`boot`。

由于 `route` 的注册工作由之前 `RoutingServiceProvider` 完成，所以 `RouteServiceProvider` 的 `register` 是空的，这里它只负责路由的启动与加载工作，我们主要看 `boot`：

```
namespace Illuminate\Foundation\Support\Providers;

class RouteServiceProvider extends ServiceProvider
{
    public function register()
    {
        //
    }

    public function boot()
    {
        $this->setRootControllerNamespace();

        if ($this->app->routesAreCached()) {
            $this->loadCachedRoutes();
        } else {
            $this->loadRoutes();

            $this->app->booted(function () {
                $this->app['router']->getRoutes()->refreshNameLo
okups();
                $this->app['router']->getRoutes()->refreshAction
Lookups();
            });
        }
    }

    protected function loadCachedRoutes()
    {
        $this->app->booted(function () {
            require $this->app->getCachedRoutesPath();
        });
    }

    protected function loadRoutes()
    {
        if (method_exists($this, 'map')) {
            $this->app->call([$this, 'map']);
        }
```

```
    }
}

class Application extends Container implements ApplicationContra
ct, HttpKernelInterface
{
    public function routesAreCached()
    {
        return $this['files']->exists($this->getCachedRoutesPath
());
    }

    public function getCachedRoutesPath()
    {
        return $this->bootstrapPath().'/cache/routes.php';
    }
}
```

从 `boot` 中可以看出，`laravel` 首先去寻找路由的缓存文件，没有缓存文件再去进行加载路由。缓存文件一般在 `bootstrap/cache/routes.php` 文件中。

加载路由主要调用 `map` 函数，这个函数一般在 `App\Providers\RouteServiceProvider` 这个类中，这个类继承上面的 `Illuminate\Foundation\Support\Providers\RouteServiceProvider`：

```
use Illuminate\Foundation\Support\Providers\RouteServiceProvider
as ServiceProvider;

class RouteServiceProvider extends ServiceProvider
{
    public function map()
    {
        $this->mapApiRoutes();

        $this->mapWebRoutes();

        //
    }

    protected function mapWebRoutes()
    {
        Route::middleware('web')
             ->namespace($this->namespace)
             ->group(base_path('routes/web.php'));
    }

    protected function mapApiRoutes()
    {
        Route::prefix('api')
             ->middleware('api')
             ->namespace($this->namespace)
             ->group(base_path('routes/api.php'));
    }
}
```

`laravle` 将路由分为两个大组：`api` 、 `web` 。这两个部分的路由分别写在两个文件中： `routes/web.php` 、 `routes/api.php` 。

路由的加载

所谓的路由加载，就是将定义路由时添加的属性，例如 'name'、'domain'、'scheme' 等等保存起来，以待后用。

- `laravel` 定义路由的属性的方法很灵活，可以定义在路由群组前，例如：

```
Route::domain('route.domain.name')
     ->group(function() {
            Route::get('foo','controller@method');
     })
```

- 可以定义在路由群组中，例如：

```
Route::group('domain' => 'group.domain.name',function() {
            Route::get('foo','controller@method');
    })
```

- 可以定义在 `method` 的前面，例如：

```
Route::domain('route.domain.name')
     ->get('foo','controller@method');
```

- 可以定义在 `method` 中，例如：

```
Route::get('foo',['domain' => 'route.domain.name','use' => 'cont
roller@method']);
```

- 还可以定义在 `method` 后，例如：

```
 Route::get('{one}', 'use' => 'controller@method')
      ->where('one', '(.+)');
```

事实上，路由的加载功能主要有三个类负责：`Illuminate\Routing\Router`、`Illuminate\Routing\Route`、`Illuminate\Routing\RouteRegistrar`。

`Router` 在整个路由功能中都是起着中枢的作用，`RouteRegistrar` 主要负责位于 `group`、`method` 这些函数之前的属性注册，例如上面的第一种和第三种，`route` 主要负责位于 `group`、`method` 这些函数之后的属性注册，例如第五种。

RouteRegistrar 路由加载

属性注册

当我们想要在 `Route` 后面直接利用 `domain()` 、 `name()` 等函数来为路由注册属性的时候，我们实际调用的是 `router` 的魔术方法 `__call()` ：

```
namespace Illuminate\Routing;

class Router implements RegistrarContract, BindingRegistrar
{
    public function __call($method, $parameters)
    {
        if (static::hasMacro($method)) {
            return $this->macroCall($method, $parameters);
        }

        return (new RouteRegistrar($this))->attribute($method, $
parameters[0]);
    }
}
```

在类 `RouteRegistrar` 中：

```
class RouteRegistrar
{
    protected $allowedAttributes = [
        'as', 'domain', 'middleware', 'name', 'namespace', 'pref
ix',
    ];

    public function attribute($key, $value)
    {
        if (! in_array($key, $this->allowedAttributes)) {
            throw new InvalidArgumentException("Attribute [{$key
}] does not exist.");
        }

        $this->attributes[array_get($this->aliases, $key, $key)]
 = $value;

        return $this;
    }
}
```

添加路由

注册属性之后，创建路由的时候，可以仅仅提供 `uri`，可以提供 `uri` 与闭包，可以提供 `uri` 与控制器，可以提供 `uri` 与数组：

```
Route::as('Foo')
     ->namespace('Namespace\\Example\\')
     ->get('foo/bar');//仅仅 uri

Route::as('Foo')
     ->namespace('Namespace\\Example\\')
     ->get('foo/bar', function () {
       }); //uri 与闭包

Route::as('Foo')
     ->namespace('Namespace\\Example\\')
     ->get('foo/bar', 'controller@method');//uri 与控制器

Route::as('Foo')
     ->namespace('Namespace\\Example\\')
     ->get('foo/bar', ['as'=> 'foo','use' =>'controller@method']
);//uri 与数组
```

利用 `get` 、 `post` 等方法创建新的路由时，会调用类 `RouteRegistrar` 中的魔术方法 `__call()` ：

```
class RouteRegistrar
{
    protected $passthru = [
        'get', 'post', 'put', 'patch', 'delete', 'options', 'any'
,
    ];

    public function __call($method, $parameters)
    {
        if (in_array($method, $this->passthru)) {
            return $this->registerRoute($method, ...$parameters)
;
        }

        if (in_array($method, $this->allowedAttributes)) {
            return $this->attribute($method, $parameters[0]);
        }
```

```
        throw new BadMethodCallException("Method [{$method}] doe
s not exist.");
    }

    protected function registerRoute($method, $uri, $action = nu
ll)
    {
        if (! is_array($action)) {
            $action = array_merge($this->attributes, $action ? [
'uses' => $action] : []);
        }

        return $this->router->{$method}($uri, $this->compileActi
on($action));
    }

    protected function compileAction($action)
    {
        if (is_null($action)) {
            return $this->attributes;
        }

        if (is_string($action) || $action instanceof Closure) {
            $action = ['uses' => $action];
        }

        return array_merge($this->attributes, $action);
    }
}
```

也就是说， `RouteRegistrar` 在这里会为闭包或控制器等所有非数组的 `action` 添加 `use` 键，然后才会去 `router` 中创建路由。

添加路由群组

注册属性之后，还可以创建路由群组，但是这时路由群组不允许添加属性 `action` ：

```
class RouteRegistrar
{
    public function group($callback)
    {
        $this->router->group($this->attributes, $callback);
    }
}
```

Router 路由群组加载

路由群组的功能可以不断叠加递归，因此每次调用 `group` ，都要用新路由群组的属性与旧路由群组属性合并，以待新的路由去继承。 `group` 参数可以是闭包函数，也可以是包含定义路由的文件路径。

```
public function group(array $attributes, $routes)
{
    $this->updateGroupStack($attributes);

    $this->loadRoutes($routes);

    array_pop($this->groupStack);
}

protected function updateGroupStack(array $attributes)
{
    if (! empty($this->groupStack)) {
        $attributes = RouteGroup::merge($attributes, end($this->
groupStack));
    }

    $this->groupStack[] = $attributes;
}

protected function loadRoutes($routes)
{
    if ($routes instanceof Closure) {
        $routes($this);
    } else {
        $router = $this;

        require $routes;
    }
}
```

关于路由群组属性的合并,

- `prefix` 、 `as` 、 `namespace` 这几个属性会连接在一起，例如 `prefix1/prefix2/prefix3` ，
- `where` 属性数组相同的会被替换，不同的会被合并。
- `domain` 属性会被替换。
- 其他属性，例如 `middleware` 数组会直接被合并，即使存在相同的元素。

```
class RouteGroup
{
    public static function merge($new, $old)
    {
        if (isset($new['domain'])) {
            unset($old['domain']);
        }

        $new = array_merge(static::formatAs($new, $old), [
            'namespace' => static::formatNamespace($new, $old),
            'prefix' => static::formatPrefix($new, $old),
            'where' => static::formatWhere($new, $old),
        ]);

        return array_merge_recursive(Arr::except(
            $old, ['namespace', 'prefix', 'where', 'as']
        ), $new);
    }
}
```

Router 路由加载

添加路由需要很多步骤，需要将路由本身的属性和路由群组的属性相结合。

```
public function get($uri, $action = null)
{
    return $this->addRoute(['GET', 'HEAD'], $uri, $action);
}

protected function addRoute($methods, $uri, $action)
{
    return $this->routes->add($this->createRoute($methods, $uri,
 $action));
}

protected function createRoute($methods, $uri, $action)
{

    if ($this->actionReferencesController($action)) {
        $action = $this->convertToControllerAction($action);
    }

    $route = $this->newRoute(
        $methods, $this->prefix($uri), $action
    );

    if ($this->hasGroupStack()) {
        $this->mergeGroupAttributesIntoRoute($route);
    }

    $this->addWhereClausesToRoute($route);

    return $route;
}
```

从上面来看，添加一个新的路由需要：

- 给路由的控制器添加 `group` 的 `namespace`
- 给路由的 `uri` 添加 `group` 的 `prefix` 前缀
- 创建新的路由
- 更新路由的属性信息
- 为路由添加 `router` - `pattern` 正则约束
- 路由添加到 `RouteCollection` 中

控制器 namespace

路由控制器的命名空间一般不用特别指定，默认值是 `\App\Http\Controllers` ，每次创建新的路由，都要将默认的命名空间添加到控制器中去：

```
protected function actionReferencesController($action)
{
    if (! $action instanceof Closure) {
        return is_string($action) || (isset($action['uses']) &&
is_string($action['uses']));
    }

    return false;
}

protected function convertToControllerAction($action)
{
    if (is_string($action)) {
        $action = ['uses' => $action];
    }

    if (! empty($this->groupStack)) {
        $action['uses'] = $this->prependGroupNamespace($action['
uses']);
    }

    $action['controller'] = $action['uses'];

    return $action;
}

protected function prependGroupNamespace($class)
{
    $group = end($this->groupStack);

    return isset($group['namespace']) && strpos($class, '\\') !=
= 0
            ? $group['namespace'].'\\'.$class : $class;
}
```

uri 前缀

在创建新的路由前，需要将路由群组的 `prefix` 添加到路由的 `uri` 中：

```
protected function prefix($uri)
{
    return trim(trim($this->getLastGroupPrefix(), '/').'/'.trim(
$uri, '/'), '/') ?: '/';
}

public function getLastGroupPrefix()
{
    if (! empty($this->groupStack)) {
        $last = end($this->groupStack);

        return isset($last['prefix']) ? $last['prefix'] : '';
    }

    return '';
}
```

创建新的路由

路由的创建需要 `Route` 类：

```
protected function newRoute($methods, $uri, $action)
{
    return (new Route($methods, $uri, $action))
                ->setRouter($this)
                ->setContainer($this->container);
}
```

关于 `Router` 类添加新的路由我们在下一部分详细说。

更新路由属性信息

创建新的路由之后，需要将路由本身的属性 `action` 与路由群组的属性结合在一起：

```
public function hasGroupStack()
{
    return ! empty($this->groupStack);
}

protected function mergeGroupAttributesIntoRoute($route)
{
    $route->setAction($this->mergeWithLastGroup($route->getActio
n()));
}
```

添加全局正则约束到路由

上一篇文章我们说过，我们可以为路由通过 `pattern` 方法添加全局的参数正则约束，所有每次添加新的路由都要将这个全局正则约束添加到路由中：

```
public function pattern($key, $pattern)
{
    $this->patterns[$key] = $pattern;
}

protected function addWhereClausesToRoute($route)
{
    $route->where(array_merge(
        $this->patterns, isset($route->getAction()['where']) ? $
route->getAction()['where'] : []
    ));

    return $route;
}
```

Route 路由加载

前面说过，路由的创建是由 `Route` 这个类完成的：

```
 public function __construct($methods, $uri, $action)
{
    $this->uri = $uri;
    $this->methods = (array) $methods;
    $this->action = $this->parseAction($action);

    if (in_array('GET', $this->methods) && ! in_array('HEAD', $t
his->methods)) {
        $this->methods[] = 'HEAD';
    }

    if (isset($this->action['prefix'])) {
        $this->prefix($this->action['prefix']);
    }
}
```

由此可以看出，路由的创建主要是路由的各个属性的初始化，其中值得注意的有两个： `action` 与 `prefix`

action 解析

```
protected function parseAction($action)
{
    return RouteAction::parse($this->uri, $action);
}
```

我们可以看出，添加新的路由时， `action` 属性需要利用 `RouteAction` 类：

```
class RouteAction
{
    public static function parse($uri, $action)
    {
        if (is_null($action)) {
            return static::missingAction($uri);
        }

        if (is_callable($action)) {
```

```
            return ['uses' => $action];
        }

        elseif (! isset($action['uses'])) {
            $action['uses'] = static::findCallable($action);
        }

        if (is_string($action['uses']) && ! Str::contains($actio
n['uses'], '@')) {
            $action['uses'] = static::makeInvokable($action['use
s']);
        }

        return $action;
    }

    protected static function findCallable(array $action)
    {
        return Arr::first($action, function ($value, $key) {
            return is_callable($value) && is_numeric($key);
        });
    }

    protected static function makeInvokable($action)
    {
        if (! method_exists($action, '__invoke')) {
            throw new UnexpectedValueException("Invalid route ac
tion: [{$action}].");
        }

        return $action.'@__invoke';
    }
}
```

前面的博客我们说过，创建路由的时候，除了为路由分配控制器之外，还可以为路由分配闭包函数，还有类函数，例如之前说的单动作控制器：

```
$router->get('foo/bar2', ['domain' => 'www.example.com', 'Illuminate\Tests\Routing\ActionStub']);

class ActionStub
{
    public function __invoke()
    {
        return 'hello';
    }
}
```

因此，解析 `action` 主要做两件事：

- 为闭包函数添加 `use` 键。对于此时没有 `use` 键的路由，由于之前在 `Router` 中已经为控制器添加 `use` 键，因此这时没有 `use` 键的，必然是闭包函数，在这里直接或者在 `action` 中寻找闭包函数后，为闭包函数添加 `use` 键。
- 单动作控制器添加 `__invoke` 。对于单动作控制器来说，此时已经和控制器一样拥有 'use' 键，但是并没有 `@` 符号，此时就会调用 `makeInvokable` 函数来将 `__invoke` 添加到后面。

prefix 前缀

路由自身也有 `prefix` 属性，而且这个属性要加在其他 `prefix` 的最前面,作为路由的 `uri` ：

```
public function prefix($prefix)
{
    $uri = rtrim($prefix, '/').'/'.ltrim($this->uri, '/');

    $this->uri = trim($uri, '/');

    return $this;
}
```

Route 路由属性加载

除了 `RouteRegistrar` 之外，`Route` 也可以为路由添加属性：

prefix 前缀

```
public function prefix($prefix)
{
    $uri = rtrim($prefix, '/').'/'.ltrim($this->uri, '/');

    $this->uri = trim($uri, '/');

    return $this;
}
```

where 正则约束

```
public function where($name, $expression = null)
{
    foreach ($this->parseWhere($name, $expression) as $name => $
expression) {
        $this->wheres[$name] = $expression;
    }

     return $this;
}

protected function parseWhere($name, $expression)
{
    return is_array($name) ? $name : [$name => $expression];
}
```

middleware 中间件

```
public function middleware($middleware = null)
{
    if (is_null($middleware)) {
        return (array) Arr::get($this->action, 'middleware', [])
;
    }

    if (is_string($middleware)) {
        $middleware = func_get_args();
    }

    $this->action['middleware'] = array_merge(
        (array) Arr::get($this->action, 'middleware', []), $midd
leware
    );

    return $this;
}
```

uses 控制器

```
public function uses($action)
{
    $action = is_string($action) ? $this->addGroupNamespaceToStr
ingUses($action) : $action;

    return $this->setAction(array_merge($this->action, $this->pa
rseAction([
        'uses' => $action,
        'controller' => $action,
    ])));
}
```

name 命名

```
public function name($name)
{
    $this->action['as'] = isset($this->action['as']) ? $this->ac
tion['as'].$name : $name;

    return $this;
}
```

RouteCollection 添加路由

在上面的部分，我们看到添加路由的代码：

```
protected function addRoute($methods, $uri, $action)
{
    return $this->routes->add($this->createRoute($methods, $uri,
 $action));
}
```

新创建的路由会加入到 `RouteCollection` 中，会更新类中的 `routes` 、 `allRoutes` 、 `nameList` 、 `actionList` 。

```
public function add(Route $route)
{
    $this->addToCollections($route);

    $this->addLookups($route);

    return $route;
}

protected function addToCollections($route)
{
    $domainAndUri = $route->domain().$route->uri();

    foreach ($route->methods() as $method) {
        $this->routes[$method][$domainAndUri] = $route;
    }

    $this->allRoutes[$method.$domainAndUri] = $route;
}

protected function addLookups($route)
{
    $action = $route->getAction();

    if (isset($action['as'])) {
        $this->nameList[$action['as']] = $route;
    }

   if (isset($action['controller'])) {
        $this->addToActionList($action, $route);
    }
}

protected function addToActionList($action, $route)
{
    $this->actionList[trim($action['controller'], '\\')] = $rout
e;
}
```

我们在上面路由的注册启动章节说道，路由的启动是 `namespace Illuminate\Foundation\Support\Providers\RouteServiceProvider` 完成的，调用的是 `boot` 函数：

```
public function boot()
{
    $this->setRootControllerNamespace();

    if ($this->app->routesAreCached()) {
        $this->loadCachedRoutes();
    } else {
        $this->loadRoutes();

        $this->app->booted(function () {
            $this->app['router']->getRoutes()->refreshNameLookup
s();
        });
    }
}
```

在最后一句，程序将会在所有服务都启动后运行 `refreshNameLookups` 函数，把所有的 `name` 属性加载到 `RouteCollection` 中:

```
public function refreshNameLookups()
{
    $this->nameList = [];

    foreach ($this->allRoutes as $route) {
        if ($route->getName()) {
            $this->nameList[$route->getName()] = $route;
        }
    }
}
```

测试样例如下：

```
public function testRouteCollectionCanRefreshNameLookups()
{
    $routeIndex = new Route('GET', 'foo/index', [
        'uses' => 'FooController@index',
    ]);

    $this->assertNull($routeIndex->getName());

    $this->routeCollection->add($routeIndex)->name('route_name')
;

    $this->assertNull($this->routeCollection->getByName('route_n
ame'));

    $this->routeCollection->refreshNameLookups();
    $this->assertEquals($routeIndex, $this->routeCollection->get
ByName('route_name'));
}
```

前言

当所有的路由都加载完毕后，就会根据请求的 `url` 来将请求分发到对应的路由上去。然而，在分发到路由之前还要经过各种中间件的计算。 `laravel` 利用装饰者模式来实现中间件的功能。

从原始装饰者模式到闭包装饰者

装饰者模式是设计模式的一种，主要进行对象的多次处理与过滤，是在开放-关闭原则下实现动态添加或减少功能的一种方式。下面先看一个装饰者模式的例子：

总共有两种咖啡：Decaf、Espresso，另有两种调味品：Mocha、Whip（3种设计的主要差别在于抽象方式不同）

装饰模式分为3个部分：

1，抽象组件 -- 对应Coffee类

2，具体组件 -- 对应具体的咖啡，如：Decaf，Espresso

3，装饰者 -- 对应调味品，如：Mocha，Whip

原始装饰者模式

```
public interface Coffee
{
    public double cost();
}

public class Espresso implements Coffee
{
    public double cost()
    {
        return 2.5;
    }
}
```

```
public class Dressing implements Coffee
{
    private Coffee coffee;

    public Dressing(Coffee coffee)
    {
        this.coffee = coffee;
    }

    public double cost()
    {
        return coffee.cost();
    }
}

public class Whip extends Dressing {
    public Whip(Coffee coffee)
    {
        super(coffee);
    }

    public double cost()
    {
        return super.cost() + 0.1;
    }
}

public class Mocha extends Dressing
{
    public Mocha(Coffee coffee)
    {
        super(coffee);
    }

    public double cost()
    {
        return super.cost() + 0.5;
    }
}
```

当我们使用装饰者模式的时候：

```
public class Test {
    public static void main(String[] args) {
        Coffee coffee = new Espresso();
        coffee = new Mocha(coffee);
        coffee = new Mocha(coffee);
        coffee = new Whip(coffee);
        //3.6(2.5 + 0.5 + 0.5 + 0.1)
        System.out.println(coffee.cost());
    }
}
```

我们可以看出来，装饰者模式就是利用装饰者类来对具体类不断的进行多层次的处理，首先我们创建了 `Espresso` 类，然后第一次利用 `Mocha` 装饰者对 `Espresso` 咖啡加了摩卡，第二次重复加了摩卡，第三次利用装饰者 `Whip` 对 `Espresso` 咖啡加了奶油。每次加入新的调料，装饰者都会对价格 `cost` 做一些处理（+0.1、+0.5）。

无构造函数的装饰者

我们对这个装饰者进行一些改造：

```
public class Espresso
{
    double cost;

    public double cost()
    {
        $this-> cost = 2.5;
    }
}

public class Dressing
{
    public double cost(Espresso $espresso)
    {
        return ($espresso);
    }
}

public class Whip extends Dressing
{
    public double cost(Espresso $espresso)
    {
        $espresso->cost = espresso->cost() + 0.1;

        return ($espresso);
    }
}

public class Mocha extends Dressing
{
    public double cost(Espresso $espresso)
    {
        $espresso->cost = espresso->cost() + 0.5;

        return ($espresso);
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Coffee $coffee = new Espresso();

        $coffee = (new Mocha())->cost($coffee);
        $coffee = (new Mocha())->cost($coffee);
        $coffee = (new Whip())->cost($coffee);

        //3.6(2.5 + 0.5 + 0.5 + 0.1)
        System.out.println(coffee.cost());
    }
}
```

改造后，装饰者类通过函数 `cost` 来注入具体类 `caffee` ，而不是通过构造函数，这样做有助于自动化进行装饰处理。我们改造后发现，想要对具体类通过装饰类进行处理，需要不断的调用 `cost` 函数，如果有10个装饰操作，就要手动写10个语句，因此我们继续进行改造：

闭包装饰者模式

```
public class Espresso
{
    double cost;

    public double cost()
    {
        $this-> cost = 2.5;
    }
}

public class Dressing
{
    public double cost(Espresso $espresso,  Closure $closure)
    {
        return ($espresso);
    }
}

public class Whip extends Dressing
{
    public double cost(Espresso $espresso, Closure $closure)
    {
        $espresso->cost = espresso->cost() + 0.1;

        return $closure($espresso);
    }
}

public class Mocha extends Dressing
{
    public double cost(Espresso $espresso, Closure $closure)
    {
        $espresso->cost = espresso->cost() + 0.5;

        return $closure($espresso);
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Coffee $coffee = new Espresso();

        $fun = function($coffee，$fuc，$dressing) {
            $dressing->cost($coffee, $fuc);
        }


        $fuc0 = function($coffee) {
            return $coffee;
        };

        $fuc1 = function($coffee) use ($fuc0, $dressing = (new Mocha()，$fun)) {
            return $fun($coffee, $fuc0, $dressing);
        }

        $fuc2 = function($coffee) use ($fuc1, $dressing = (new Mocha()，$fun)) {
            return $fuc($coffee, $fun1, $dressing);
        }

        $fuc3 = function($coffee) use ($fuc2, $dressing = (new Whip()，$fun)) {
            return $fuc($coffee, $fun2, $dressing);
        }

        $coffee = $fun3($coffee);

        //3.6(2.5 + 0.5 + 0.5 + 0.1)
        System.out.println(coffee.cost());
    }
}
```

在这次改造中，我们使用了闭包函数，这样做的目的在于，我们只需要最后一句 `$fun3($coffee)` ,就可以启动整个装饰链条。

闭包装饰者的抽象化

然而这种改造还不够深入，因为我们还可以把 `$fuc1` 、 `$fuc2` 、 `$fuc3` 继续抽象化为一个闭包函数，这个闭包函数仅仅是参数 `$fuc` 、 `$dressing` 每次不同， `$coffee` 相同，因此改造如下：

```
public class Test {
    public static void main(String[] args) {
        Coffee $coffee = new Espresso();

        $fun = function($coffee) use ($fuc，$dressing) {
            $dressing->cost($coffee, $fuc);
        }

        $fuc = function($fuc，$dressing) use ($fun) {
            return $fun;
        };


        $fuc0 = function($coffee) {
            return $coffee;
        };

        $fuc1 = $fuc($fuc0, (new Mocha());

        $fuc2 = $fuc($fuc1, (new Mocha());

        $fuc3 = $fuc($fuc2, (new Whip());

        $coffee = $fun3($coffee);

        //3.6(2.5 + 0.5 + 0.5 + 0.1)
        System.out.println(coffee.cost());
    }
}
```

这次，我们把之前的闭包分为两个部分， `$fun` 负责具体类的参数传递， `$fuc` 负责装饰者和闭包函数的参数传递。在最后一句 `$fun3` ,只需要传递一个具体类，就可以启动整个装饰链条。

闭包装饰者的自动化

到这里，我们还有一件事没有完成，那就是 `$fuc1` 、 `$fuc2` 、 `$fuc3` 这些闭包的构建还是手动的，我们需要将这个过程改为自动的：

```
public class Test {
    public static void main(String[] args) {
        Coffee $coffee = new Espresso();

        $fun = function($coffee) use ($fuc，$dressing) {
            $dressing->cost($coffee, $fuc);
        }

        $fuc = function($fuc，$dressing) use ($fun) {
            return $fun;
        };


        $fuc0 = function($coffee) {
            return $coffee;
        };

        $fucn = array_reduce(
            [(new Mocha(),(new Mocha(),(new Whip()], $fuc, $fuc0
        );

        $coffee = $fucn($coffee);

        //3.6(2.5 + 0.5 + 0.5 + 0.1)
        System.out.println(coffee.cost());
    }
}
```

laravel的闭包装饰者——Pipeline

上一章我们说到了路由的注册启动与加载过程，这个过程由 `bootstrap()` 完成。当所有的路由加载完毕后，就要进行各种中间件的处理了：

```
protected function sendRequestThroughRouter($request)
{
    $this->app->instance('request', $request);

    Facade::clearResolvedInstance('request');

    $this->bootstrap();

    return (new Pipeline($this->app))
                ->send($request)
                ->through($this->app->shouldSkipMiddleware() ? [
] : $this->middleware)
                ->then($this->dispatchToRouter());
}

public function shouldSkipMiddleware()
{
    return $this->bound('middleware.disable') &&
           $this->make('middleware.disable') === true;
}
```

`laravel` 的中间件处理由 `Pipeline` 来完成，它是一个闭包装饰者模式，其中

- `request` 是具体类，相当于我们上面的 `caffee` 类；
- `middleware` 中间件是装饰者类，相当于上面的 `dressing` 类；

我们先看看这个类内部的代码：

```
class Pipeline implements PipelineContract
{
    public function __construct(Container $container = null)
    {
      $this->container = $container;
    }
```

```
    public function send($passable)
    {
        $this->passable = $passable;

        return $this;
    }

    public function through($pipes)
    {
        $this->pipes = is_array($pipes) ? $pipes : func_get_args
();

        return $this;
    }

    public function then(Closure $destination)
    {
        $pipeline = array_reduce(
            array_reverse($this->pipes), $this->carry(), $this->
prepareDestination($destination)
        );

        return $pipeline($this->passable);
    }

    protected function prepareDestination(Closure $destination)
    {
        return function ($passable) use ($destination) {
            return $destination($passable);
        };
    }

    protected function carry()
    {
        return function ($stack, $pipe) {
            return function ($passable) use ($stack, $pipe) {
                if ($pipe instanceof Closure) {
                    return $pipe($passable, $stack);
                } elseif (! is_object($pipe)) {
```

```
                    list($name, $parameters) = $this->parsePipeS
tring($pipe);

                    $pipe = $this->getContainer()->make($name);

                    $parameters = array_merge([$passable, $stack
], $parameters);
                } else {
                    $parameters = [$passable, $stack];
                }

                return $pipe->{$this->method}(...$parameters);
            };
        };
    }
}
```

`pipeline` 的构造和我们上面所讲的闭包装饰者相同，我们着重来看 `carry()` 函数的代码：

```
function ($stack, $pipe) {
    ...
}
```

最外层的闭包相当于上个章节的 `$fuc` ,

```
function ($passable) use ($stack, $pipe) {
    ...
}
```

里面的这一层比闭包型党与上个章节的 `$fun` ，

`prepareDestination` 这个函数相当于上面的 `$fuc0` ,

```
            if ($pipe instanceof Closure) {
                return $pipe($passable, $stack);
            } elseif (! is_object($pipe)) {
                list($name, $parameters) = $this->parsePipeStrin
g($pipe);

                $pipe = $this->getContainer()->make($name);

                $parameters = array_merge([$passable, $stack], $
parameters);
            } else {
                $parameters = [$passable, $stack];
            }

            return $pipe->{$this->method}(...$parameters);
```

这一部分相当于上个章节的 `$dressing->cost($coffee, $fuc);` ,这部分主要解析中间件 `handle()` 函数的参数：

```
public function via($method)
{
    $this->method = $method;

    return $this;
}

protected function parsePipeString($pipe)
{
    list($name, $parameters) = array_pad(explode(':', $pipe, 2),
2, []);

    if (is_string($parameters)) {
        $parameters = explode(',', $parameters);
    }

    return [$name, $parameters];
}
```

这样，`laravel` 就实现了中间件对 `request` 的层层处理。

前言

利用 `pipeline` 进行中间件的层层处理后，接下来 `laravel` 就会利用请求的 `url` 来寻找与其对应的路由， `laravel` 采用对路由注册的 `uri` 进行正则编译，然后利用 `request` 的 `url` 进行正则匹配来寻找正确的路由。

前期准备

在上一篇文章中，我们了解了 `Pipeline` 的原理，我们知道它调用了 `dispatchToRouter()` 这个函数：

```
protected function sendRequestThroughRouter($request)
{
    $this->app->instance('request', $request);

    Facade::clearResolvedInstance('request');

    $this->bootstrap();

    return (new Pipeline($this->app))
                ->send($request)
                ->through($this->app->shouldSkipMiddleware() ? [
] : $this->middleware)
                ->then($this->dispatchToRouter());
}

protected function dispatchToRouter()
{
    return function ($request) {
        $this->app->instance('request', $request);

        return $this->router->dispatch($request);
    };
}
```

这个函数实际上利用的是 `Router` 的 `dispatch`,这个函数的任务是进行路由匹配，并且调用路由绑定的控制器或者闭包函数：

```
class Router implements RegistrarContract, BindingRegistrar
{
    public function dispatch(Request $request)
    {
        $this->currentRequest = $request;

        return $this->dispatchToRoute($request);
    }

    public function dispatchToRoute(Request $request)
    {
        $route = $this->findRoute($request);

        $request->setRouteResolver(function () use ($route) {
            return $route;
        });

        $this->events->dispatch(new Events\RouteMatched($route,
$request));

        $response = $this->runRouteWithinStack($route, $request)
;

        return $this->prepareResponse($request, $response);
    }
}
```

我们这篇文章就是讲解第一句: `findRoute()` 路由匹配:

```
protected function findRoute($request)
{
    $this->current = $route = $this->routes->match($request);

    $this->container->instance(Route::class, $route);

    return $route;
}
```

寻找路由的任务由 `RouteCollection` 负责，这个函数负责匹配路由，并且把 `request` 的 `url` 参数绑定到路由中：

```
class RouteCollection implements Countable, IteratorAggregate
{
    public function match(Request $request)
    {
        $routes = $this->get($request->getMethod());

        $route = $this->matchAgainstRoutes($routes, $request);

        if (! is_null($route)) {
            return $route->bind($request);
        }

        $others = $this->checkForAlternateVerbs($request);

        if (count($others) > 0) {
            return $this->getRouteForMethods($request, $others);
        }

        throw new NotFoundHttpException;
    }

    protected function matchAgainstRoutes(array $routes, $reques
t, $includingMethod = true)
    {
        return Arr::first($routes, function ($value) use ($reque
st, $includingMethod) {
            return $value->matches($request, $includingMethod);
        });
    }
}
```

路由正则匹配

如何去寻找请求 `request` 想要调用的路由呢？ `laravel` 首先对路由进行正则编译，得到路由的正则匹配串，然后利用请求的 `url` 尝试去匹配，如果匹配成功，那么就会选定该路由：

```
class Route
{
    public function matches(Request $request, $includingMethod =
 true)
    {
        $this->compileRoute();

        foreach ($this->getValidators() as $validator) {
            if (! $includingMethod && $validator instanceof Meth
odValidator) {
                continue;
            }

            if (! $validator->matches($this, $request)) {
                return false;
            }
        }

        return true;
    }

    protected function compileRoute()
    {
        if (! $this->compiled) {
            $this->compiled = (new RouteCompiler($this))->compil
e();
        }

       return $this->compiled;
    }
}
```

可以看出，路由的正则编译由 `RouteCompiler` 类专门负责：

```
class RouteCompiler
{
     public function __construct($route)
    {
        $this->route = $route;
    }

    public function compile()
    {
        $optionals = $this->getOptionalParameters();

        $uri = preg_replace('/\{(\w+?)\?\}/', '{$1}', $this->rou
te->uri());

        return (
            new SymfonyRoute($uri, $optionals, $this->route->whe
res, [], $this->route->domain() ?: '')
        )->compile();
    }
}
```

可以看出，`laravel` 真正的正则编译是重用 `symfony` 框架的，但是在利用 `symfony` 进行正则编译之前，`laravel` 先对路由的 `uri` 进行了一些处理，以适应 `symfony` 的要求。

路由可选参数转换

对于 `laravel` 来说，可以选择某个路由 `url` 的参数是可选的，通常来说，这种可选参数都有默认值。`laravel` 利用 `?` 来表示可选参数：

```
$router->get('{foo?}/{baz?}', function ($name = 'taylor', $age =
25) {
     return $name.$age;
});
```

但是对于 `symfony` 来说， `?` 没有任何特殊意义， `symfony` 利用 `SymfonyRoute` 类进行路由初始化，并把第二个参数作为可选参数，因此 `laravel` 需要把可选参数提取出来，然后赋给 `SymfonyRoute` 构造函数。

可选参数的提取由 `getOptionalParameters` 负责：

```
 protected function getOptionalParameters()
{
    preg_match_all('/\{(\w+?)\?\}/', $this->route->uri(), $match
es);

    return isset($matches[1]) ? array_fill_keys($matches[1], null
) : [];
}
```

> `preg_match_all` 函数用于进行正则表达式全局匹配，成功返回整个模式匹配的次数（可能为零），如果出错返回 FALSE。
>
> 默认排序方式为 `PREG_PATTERN_ORDER`,结果排序为 `$matches[0]` 保存完整模式的所有匹配, `$matches[1]` 保存第一个子组的所有匹配，以此类推。
>
> 若排序方式为 `PREG_SET_ORDER`,结果排序为 `$matches[0]` 包含第一次匹配得到的所有匹配(包含子组)， `$matches[1]` 是包含第二次匹配到的所有匹配(包含子组)的数组，以此类推。

以 `{foo?}/{baz?}` 为例，得到的 `matches[0]` :

```
matches[0] = array (
    0 = '{foo?}',
    1 = '{baz?}',
)
```

得到的结果 `matches` 中 `matches[1]` 是被匹配上的字符串，以 `{foo?}/{baz?}` 为例，得到的 `matches[1]` :

```
matches[1] = array (
    0 = 'foo',
    1 = 'baz',
)
```

`array_fill_keys` 函数负责使用指定的键和值填充数组，例如上例中就可以得到：

```
optionals = array (
    foo = null,
    baz = null,
)
```

得到可选参数的数组 `optionals` 后，就要将路由的 `uri` 中 `?` 替换掉，这也就是 `preg_replace` 的作用，以 `{foo?}/{baz?}` 为例，最后得到的替换结果为 `{foo}/{baz}` 。

Symfony 路由初始化

在 `symfony` 的路由初始化中，由很多参数：

- path 是路由的 `uri`
- defaults 是路由可选参数
- requirements 是路由的参数正则约束
- options 路由的选项参数，例如路由正则编译类等
- host 是路由的主域
- schenes 是 web 的协议，例如 http, https
- methods 是调用的方法，例如 `get` 、 `post`
- condition

```
namespace Symfony\Component\Routing;

class Route implements \Serializable
{
    public function __construct($path, array $defaults = array()
, array $requirements = array(), array $options = array(), $host
 = '', $schemes = array(), $methods = array(), $condition = '')
    {
        $this->setPath($path);
        $this->setDefaults($defaults);
        $this->setRequirements($requirements);
        $this->setOptions($options);
        $this->setHost($host);
        $this->setSchemes($schemes);
        $this->setMethods($methods);
        $this->setCondition($condition);
    }

    public function setOptions(array $options)
    {
        $this->options = array(
            'compiler_class' => 'Symfony\\Component\\Routing\\Ro
uteCompiler',
        );

        return $this->addOptions($options);
    }
}
```

可以看出， `laravel` 初始化路由的时候，分别初始化了 `path` 、 `defaults` 、 `requirements` 、 `host` ，其余都是默认值。其中 `host` 是路由的 `domain` 去除 `http` 、 `https` 之后的主域。

```
public function domain()
{
    return isset($this->action['domain'])
            ? str_replace(['http://', 'https://'], '', $this->ac
tion['domain']) : null;
}
```

路由的正则编译

路由的编译由 `symfony` 的 `route` 类完成：

```
public function compile()
{
    if (null !== $this->compiled) {
        return $this->compiled;
    }

    $class = $this->getOption('compiler_class');

    return $this->compiled = $class::compile($this);
}
```

`compiler_class` 是初始化的时候提供的类 `Symfony\\Component\\Routing\\RouteCompiler` .

下面是就是路由编译的主要功能实现：

compile 函数

```
namespace Symfony\Component\Routing;

class RouteCompiler implements RouteCompilerInterface
{
    public static function compile(Route $route)
    {
        $hostVariables = array();
```

```
        $variables = array();
        $hostRegex = null;
        $hostTokens = array();

        if ('' !== $host = $route->getHost()) {
            $result = self::compilePattern($route, $host, true);

            $hostVariables = $result['variables'];
            $variables = $hostVariables;

            $hostTokens = $result['tokens'];
            $hostRegex = $result['regex'];
        }

        $path = $route->getPath();

        $result = self::compilePattern($route, $path, false);

        $staticPrefix = $result['staticPrefix'];

        $pathVariables = $result['variables'];

        foreach ($pathVariables as $pathParam) {
            if ('_fragment' === $pathParam) {
                throw new \InvalidArgumentException(sprintf('Rou
te pattern "%s" cannot contain "_fragment" as a path parameter.'
, $route->getPath()));
            }
        }

        $variables = array_merge($variables, $pathVariables);

        $tokens = $result['tokens'];
        $regex = $result['regex'];

        return new CompiledRoute(
            $staticPrefix,
            $regex,
            $tokens,
            $pathVariables,
```

```
                $hostRegex,
                $hostTokens,
                $hostVariables,
                array_unique($variables)
            );
        }
    }
```

可以看出，路由的正则编译由两个部分构成：主域的正则编译与 `uri` 的正则编译。这两个部分的编译功能由函数 `compilePattern` 负责，这个函数会有返回三种数据结果，以 `/foo/{bar}` 为例：

- `variables` 代表正则匹配的路由参数,如 `bar`
- `tokens` 代表正则匹配的普通路由字符串,如 `foo`
- `regex` 代表路由匹配的正则表达式结果
- 有时候也会有 `$staticPrefix` ,这个是路由 `url` 前没有路由参数的字符串前缀，如 `/foo/` .

compilePattern 函数

由于 `symfony` 原始的正则编译稍微复杂，本文剔除了一些处理 `utf8` 和异常处理的代码，特意挑选计算正则表达式的主干代码，如下：

```
    private static function compilePattern(Route $route, $patter
n, $isHost)
    {
        $tokens = array();
        $variables = array();
        $matches = array();
        $pos = 0;
        $defaultSeparator = $isHost ? '.' : '/';

        preg_match_all('#\{\w+\}#', $pattern, $matches, PREG_OFF
SET_CAPTURE | PREG_SET_ORDER);
        foreach ($matches as $match) {
            $varName = substr($match[0][0], 1, -1);
            $precedingText = substr($pattern, $pos, $match[0][1]
 - $pos);
```

```
            $pos = $match[0][1] + strlen($match[0][0]);

            if (!strlen($precedingText)) {
                $precedingChar = '';
            } else {
                $precedingChar = substr($precedingText, -1);
            }
            $isSeparator = '' !== $precedingChar && false !== st
rpos(static::SEPARATORS, $precedingChar);

            if ($isSeparator && $precedingText !== $precedingCha
r) {
                $tokens[] = array('text', substr($precedingText,
0, -strlen($precedingChar)));
            } elseif (!$isSeparator && strlen($precedingText) > 0
) {
                $tokens[] = array('text', $precedingText);
            }

            $regexp = $route->getRequirement($varName);
            if (null === $regexp) {
                $followingPattern = (string) substr($pattern, $p
os);

                $nextSeparator = self::findNextSeparator($follow
ingPattern, $useUtf8);
                $regexp = sprintf(
                    '[^%s%s]+',
                    preg_quote($defaultSeparator, self::REGEX_DE
LIMITER),
                    $defaultSeparator !== $nextSeparator && '' !
== $nextSeparator ? preg_quote($nextSeparator, self::REGEX_DELIM
ITER) : ''
                );
                if (('' !== $nextSeparator && !preg_match('#^\{\
w+\}#', $followingPattern)) || '' === $followingPattern) {
                    $regexp .= '+';
                }
            }
```

```
                $tokens[] = array('variable', $isSeparator ? $preced
ingChar : '', $regexp, $varName);
                $variables[] = $varName;
            }

            if ($pos < strlen($pattern)) {
                $tokens[] = array('text', substr($pattern, $pos));
            }

            // find the first optional token
            $firstOptional = PHP_INT_MAX;
            if (!$isHost) {
                for ($i = count($tokens) - 1; $i >= 0; --$i) {
                    $token = $tokens[$i];
                    if ('variable' === $token[0] && $route->hasDefau
lt($token[3])) {
                        $firstOptional = $i;
                    } else {
                        break;
                    }
                }
            }

            // compute the matching regexp
            $regexp = '';
            for ($i = 0, $nbToken = count($tokens); $i < $nbToken; +
+$i) {
                $regexp .= self::computeRegexp($tokens, $i, $firstOp
tional);
            }
            $regexp = self::REGEX_DELIMITER.'^'.$regexp.'$'.self::RE
GEX_DELIMITER.'s'.($isHost ? 'i' : '');

            return array(
                'staticPrefix' => 'text' === $tokens[0][0] ? $tokens[
0][1] : '',
                'regex' => $regexp,
                'tokens' => array_reverse($tokens),
                'variables' => $variables,
            );
```

```
    }
```

下面本文将以 `prefix/{foo}/{baz}.{ext}/tail` 为例，来详细讲一下路由 `uri` 的正则编译过程。

`preg_match_all` 全匹配

由于 `preg_match_all` 使用了 `PREG_SET_ORDER`,因此结果数组 `matches` 中每一个元素都是一次匹配的结果，本例中：

```
$matches = array (
    0 = array (
        0 = array (
            0 = "{foo}",
            1 = 8
        )
    )
    1 = array (
        0 = array (
            0 = "{baz}",
            1 = 14
        )
    )
    2 = array (
        0 = array (
            0 = "{ext}",
            1 = 20
        )
    )
)
```

接下来，程序会用循环来分别处理各个匹配的结果。

变量

每个匹配结果都会先计算变量:
`varName` 、 `precedingText` 、 `precedingChar` 、 `isSeparator`

- `varName` 匹配结果会将路由参数提取出来，本例中：`foo`、`baz`、`ext`
- `precedingText` 是两个路由参数之间的字符串，本例中：`prefix/`、`/`、`.`
- `precedingChar` 是每个路由参数之前的字符，也就是 `precedingText` 的最后一个字符,本例中：`/`、`/`、`.`
- `isSeparator` 判断 `precedingChar` 是否是 `url` 的间隔符，本例中：`true`、`true`、`true`

tokens-text

将 `precedingText` 记录进 `tokens` 数组，key 为 `text`。第一次循环，tokens：

```
tokens = array (
    0 = text,
    1 = prefix,
)
```

第二次循环与第三次循环由于 `precedingText` == `precedingChar`，所以并不会记录。

构建 regexp

若在路由定义的过程中利用 `where` 属性或者 `pattern` 为路由的参数设置正则约束，那么此时就会将约束规则赋给 `regexp`，否则就会启用构建 `regexp` 的过程：

```
$followingPattern = (string) substr($pattern, $pos);
$nextSeparator = self::findNextSeparator($followingPattern, $useUtf8);

$regexp = sprintf(
            '[^%s%s]+',
            preg_quote($defaultSeparator, self::REGEX_DELIMITER),
            $defaultSeparator !== $nextSeparator && '' !== $nextSeparator ? preg_quote($nextSeparator, self::REGEX_DELIMITER) : ''
);

if (('' !== $nextSeparator && !preg_match('#^\{\w+\}#', $followingPattern)) || '' === $followingPattern) {
    $regexp .= '+';
}
```

构建 `regexp` 有两个部分，

- 寻找 `nextSeparator` ：

```
private static function findNextSeparator($pattern, $useUtf8)
{
    if ('' == $pattern) {
        return '';
    }

    if ('' === $pattern = preg_replace('#\{\w+\}#', '', $pattern)) {
        return '';
    }

    return false !== strpos(static::SEPARATORS, $pattern[0]) ? $pattern[0] : '';
}
```

这个函数的意义在于为路由的 `uri` 的路由参数寻找非默认间隔符，例如，路由可以这样设置 `uri` ：

```
/{baz}.{ext}/
```

默认的间隔符就是 `/` ，如果不设置非默认间隔符的时候，那么 `regexp = [^/]` ， `mobile.html` 这样的请求就会被 `{baz}` 这个参数全部匹配到， `{ext}` 就没有任何参数来对应。设置了非默认间隔符后 `regexp = [^/.]` , `baz` 就会匹配 `mobile` ， `ext` 就会匹配 `html` 。

- 侵占型正则表达式

```
if ((''  !== $nextSeparator && !preg_match('#^\{\w+\}#', $followingPattern)) || '' === $followingPattern) {
    $regexp .= '+';
}
```

为了减少贪婪型正则表达式的回溯导致的性能浪费，当后续字符串已经结束或者不存在 `/{x}{y}` 这样情况的时候，程序将贪婪型正则表达式改为侵占型正则表达式。有关正则表达式的模式请查看：正则表达式之 贪婪与非贪婪模式详解（概述）

tokens-variable

获取路由参数和正则表达式之后，就要更新 `tokens` ,分别将 `isSeparator` , `regexp` , `varName` 更新到结果数组中。

以 `prefix/{foo}/{baz}.{ext}/tail` 为例， `$tokens` 在各个循环时值为：

```
$tokens = array (
    0 = array (
        0 = ‘text’,
        1 = '/prefix'
    )
    1 = array (
        0 = ‘variable’,
        1 = '/',
        0 = ‘[^/]++’,
        1 = 'foo'
    )//第一次循环结束
    2 = array (
        0 = ‘variable’,
        1 = '/',
        0 = ‘[^/\.]++’,
        1 = 'baz'
    )//第二次循环结束
    3 = array (
        0 = ‘variable’,
        1 = '.',
        0 = ‘[^/]++’,
        1 = 'ext'
    )//循环结束
    4 = array (
        0 = ‘text’,
        1 = '/tail'
    )//    循环外
)
```

默认路由参数

接下来就要计算首个默认路由参数在整个路由 `url` 的位置，以便在生成正则表达式中使用：

```
$firstOptional = PHP_INT_MAX;
if (!$isHost) {
    for ($i = count($tokens) - 1; $i >= 0; --$i) {
        $token = $tokens[$i];
        if ('variable' === $token[0] && $route->hasDefault($toke
n[3])) {
            $firstOptional = $i;
        } else {
            break;
        }
    }
}
```

计算正则表达式

所有的 `tokens` 数组都构建完毕，接下来就需要利用这个数组来构建正则表达式了。

```
$regexp = '';
for ($i = 0, $nbToken = count($tokens); $i < $nbToken; ++$i) {
    $regexp .= self::computeRegexp($tokens, $i, $firstOptional);
}
$regexp = self::REGEX_DELIMITER.'^'.$regexp.'$'.self::REGEX_DELI
MITER.'s'.($isHost ? 'i' : '');
```

```
    private static function computeRegexp(array $tokens, $index,
 $firstOptional)
    {
        $token = $tokens[$index];
        if ('text' === $token[0]) {
            // Text tokens
            return preg_quote($token[1], self::REGEX_DELIMITER);
        } else {
            // Variable tokens
            if (0 === $index && 0 === $firstOptional) {
                // When the only token is an optional variable t
oken, the separator is required
                return sprintf('%s(?P<%s>%s)?', preg_quote($toke
n[1], self::REGEX_DELIMITER), $token[3], $token[2]);
            } else {
                $regexp = sprintf('%s(?P<%s>%s)', preg_quote($to
ken[1], self::REGEX_DELIMITER), $token[3], $token[2]);
                if ($index >= $firstOptional) {
                    $regexp = "(?:$regexp";
                    $nbTokens = count($tokens);
                    if ($nbTokens - 1 == $index) {
                        // Close the optional subpatterns
                        $regexp .= str_repeat(')?', $nbTokens -
$firstOptional - (0 === $firstOptional ? 1 : 0));
                    }
                }

                return $regexp;
            }
        }
    }
```

`computeRegexp` 函数的大致流程为：

- 若 `tokens` 当前元素是 `text` ，不是路由参数的时候，直接赋值原字符串即可
- 若 `url` 中路由参数都是可选参数，且没有任何 `text` ，那么第一个可选参数使用捕获分组
- 若当前路由参数是可选参数的时候，需要在正则表达式中不断叠加非捕获分组

`(?` ，再最后设置为可选分组 `)?` ，例如 `(?:/(?P<baz>[^/]++)(?:/(?P<ext>[^/]++))?)?`

- 若当前路由参数不是可选参数的时候，正则表达式就是固定模式，例如：`/(?P<foo>[^/]++)`

利用 `computeRegexp` 函数拼接正则表达式后，还要在最两侧分隔符、开始符 `^`,结束符 `$` 、单行修正符 `s` ，如果是主域的正则表达式，还要添加不区分大小写的修正符 `i` 。

以 `prefix/{foo}/{baz}.{ext}/tail` 为例,每次生成的正则表达式如下：

```
/prefix
/prefix/(?P<foo>[^/]++)
/prefix/(?P<foo>[^/]++)/(?P<baz>[^/\.]++)
/prefix/(?P<foo>[^/]++)/(?P<baz>[^/\.]++)\.(?P<ext>[^/]++)
/prefix/(?P<foo>[^/]++)/(?P<baz>[^/\.]++)\.(?P<ext>[^/]++)/tail
#^/prefix/(?P<foo>[^/]++)/(?P<baz>[^/\.]++)\.(?P<ext>[^/]++)/tail$#s
```

以 `{foo?}/{baz?}.{ext?}` 为例,每次生成的正则表达式如下：

```
/(?P<foo>[^/]++)?
/(?P<foo>[^/]++)?(?:/(?P<baz>[^/\.]++)
/(?P<foo>[^/]++)?(?:/(?P<baz>[^/\.]++)(?:\.(?P<ext>[^/]++))?)?
#^/(?P<foo>[^/]++)?(?:/(?P<baz>[^/\.]++)(?:\.(?P<ext>[^/]++))?)?$#s
```

前言

上一篇文章我们说到路由的正则编译，正则编译的目的就是和请求的 `url` 来匹配，只有匹配上的路由才是我们真正想要的，此外也会通过正则匹配来获取路由的参数。

路由的匹配

路由进行正则编译后，就要与请求 `request` 来进行正则匹配，并且进行一些验证，例如 `UriValidator` 、 `MethodValidator` 、 `SchemeValidator` 、 `HostValidator` 。

```
class RouteCollection implements Countable, IteratorAggregate
{
    public function match(Request $request)
    {
        $routes = $this->get($request->getMethod());

        $route = $this->matchAgainstRoutes($routes, $request);

        if (! is_null($route)) {
            return $route->bind($request);
        }

        $others = $this->checkForAlternateVerbs($request);

        if (count($others) > 0) {
            return $this->getRouteForMethods($request, $others);
        }

        throw new NotFoundHttpException;
    }

    protected function matchAgainstRoutes(array $routes, $reques
t, $includingMethod = true)
```

```
    {
            return Arr::first($routes, function ($value) use ($r
equest, $includingMethod) {
             return $value->matches($request, $includingMethod);
            });
    }
}

class Route
{
    public function matches(Request $request, $includingMethod =
 true)
    {
            $this->compileRoute();

        foreach ($this->getValidators() as $validator) {
            if (! $includingMethod && $validator instanceof Meth
odValidator) {
                continue;
            }

            if (! $validator->matches($this, $request)) {
                return false;
            }
            }

            return true;
    }

    public static function getValidators()
    {
        if (isset(static::$validators)) {
            return static::$validators;
        }

        return static::$validators = [
            new UriValidator, new MethodValidator,
            new SchemeValidator, new HostValidator,
        ];
    }
```

```
}
```

UriValidator uri 验证

`UriValidator` 验证主要是目的是查看路由正则与请求是否匹配：

```
class UriValidator implements ValidatorInterface
{
    public function matches(Route $route, Request $request)
    {
        $path = $request->path() == '/' ? '/' : '/'.$request->pa
th();

        return preg_match($route->getCompiled()->getRegex(), raw
urldecode($path));
    }
}
```

值得注意的是，在匹配路径之前，程序使用了 `rawurldecode` 来对请求进行解码。

MethodValidator 验证

请求方法验证：

```
class MethodValidator implements ValidatorInterface
{
    public function matches(Route $route, Request $request)
    {
        return in_array($request->getMethod(), $route->methods()
);
    }
}
```

SchemeValidator 验证

路由 `scheme` 协议验证：

```
class SchemeValidator implements ValidatorInterface
{
    public function matches(Route $route, Request $request)
    {
        if ($route->httpOnly()) {
            return ! $request->secure();
        } elseif ($route->secure()) {
            return $request->secure();
        }

        return true;
    }
}

public function httpOnly()
{
    return in_array('http', $this->action, true);
}

public function secure()
{
    return in_array('https', $this->action, true);
}
```

HostValidator 验证

主域验证：

```
class HostValidator implements ValidatorInterface
{
    public function matches(Route $route, Request $request)
    {
        if (is_null($route->getCompiled()->getHostRegex())) {
            return true;
        }

        return preg_match($route->getCompiled()->getHostRegex(),
 $request->getHost());
    }
}
```

也就是说，如果路由中并不设置 `host` 属性，那么这个验证并不进行。

路由的参数绑定

一旦某个路由符合请求的 `uri` 四项认证，就将会被返回，接下来就要对路由的参数进行绑定与赋值：

```
class RouteCollection implements Countable, IteratorAggregate
{
    public function bind(Request $request)
    {
        $this->compileRoute();

        $this->parameters = (new RouteParameterBinder($this))
                        ->parameters($request);

        return $this;
    }
}
```

`bind` 函数负责路由参数与请求 `url` 的绑定工作:

```
class RouteParameterBinder
{
    public function parameters($request)
    {
        $parameters = $this->bindPathParameters($request);

        if (! is_null($this->route->compiled->getHostRegex())) {
            $parameters = $this->bindHostParameters(
                $request, $parameters
            );
        }

        return $this->replaceDefaults($parameters);
    }
}
```

可以看出，路由参数绑定分为主域参数绑定与路径参数绑定，我们先看路径参数绑定：

路径参数绑定

```
class RouteParameterBinder
{
    protected function bindPathParameters($request)
    {
            preg_match($this->route->compiled->getRegex(), '/'.$
request->decodedPath(), $matches);

            return $this->matchToKeys(array_slice($matches, 1));
    }
}
```

例如， `{foo}/{baz?}.{ext?}` 进行正则编译后结果：

```
#^/(?P<foo>[^/]++)(?:/(?P<baz>[^/\.]++)(?:\.(?P<ext>[^/]++))?)?$
#s
```

其与 `request` 匹配后的结果为：

```
$matches = array (
    0   = "/foo/baz.ext",
    1   = "foo",
    foo = "foo",
    2   = "baz",
    baz = "baz",
    3   = "ext",
    ext = "ext",
)
```

`array_slice($matches, 1)` 取出了 `$matches` 数组 1 之后的结果，然后调用了 `matchToKeys` 函数，

```
protected function matchToKeys(array $matches)
{
     if (empty($parameterNames = $this->route->parameterNames())
) {
            return [];
     }

    $parameters = array_intersect_key($matches, array_flip($para
meterNames));

    return array_filter($parameters, function ($value) {
            return is_string($value) && strlen($value) > 0;
     });
}
```

该函数中利用正则获取了路由的所有参数：

```
class Route
{
    public function parameterNames()
    {
        if (isset($this->parameterNames)) {
            return $this->parameterNames;
        }

        return $this->parameterNames = $this->compileParameterNa
mes();
    }


    protected function compileParameterNames()
    {
        preg_match_all('/\{(.*?)\}/', $this->domain().$this->uri
, $matches);

        return array_map(function ($m) {
            return trim($m, '?');
        }, $matches[1]);
    }
}
```

可以看出，获取路由参数的正则表达式采用了勉强模式，意图提取出所有的路由参数。否则，对于路由 `{foo}/{baz?}.{ext?}` ,贪婪型正则表达式 `/\{(.*)\}/` 将会匹配整个字符串，而不是各个参数分组。

提取出的参数结果为：

```
$matches = array (
    0 = array (
        0 = "{foo}".
        1 = "{baz?}",
        2 = "{ext?}",
    )
    1 = array (
        0 = "foo".
        1 = "baz?",
        2 = "ext?",
    )
)
```

得出的结果将会去除 `$matches[1]` ，并且将会删除结果中最后的 `?` 。

之后，在 `matchToKeys` 函数中，

```
$parameters = array_intersect_key($matches, array_flip($paramete
rNames));
```

获取了匹配结果与路由所有参数的交集：

```
 $parameters = array (
    foo = "foo",
    baz = "baz",
    ext = "ext",
 )
```

主域参数绑定

```
protected function bindHostParameters($request, $parameters)
{
    preg_match($this->route->compiled->getHostRegex(), $request-
>getHost(), $matches);

    return array_merge($this->matchToKeys(array_slice($matches, 1
)), $parameters);
}
```

步骤与路由参数绑定一致。

替换默认值

进行参数绑定后，有一些可选参数并没有在 `request` 中匹配到，这时候就要用可选参数的默认值添加到变量 `parameters` 中：

```
protected function replaceDefaults(array $parameters)
{
    foreach ($parameters as $key => $value) {
        $parameters[$key] = isset($value) ? $value : Arr::get($t
his->route->defaults, $key);
    }

    foreach ($this->route->defaults as $key => $value) {
        if (! isset($parameters[$key])) {
            $parameters[$key] = $value;
        }
    }

    return $parameters;
}
```

匹配异常处理

如果 `url` 匹配失败，没有找到任何路由与请求相互匹配，就会切换 `method` 方法，以求任意路由来匹配：

```
protected function checkForAlternateVerbs($request)
{
    $methods = array_diff(Router::$verbs, [$request->getMethod()
]);

    $others = [];

    foreach ($methods as $method) {
        if (! is_null($this->matchAgainstRoutes($this->get($meth
od), $request, false))) {
            $others[] = $method;
        }
    }

    return $others;
}
```

如果使用其他方法匹配成功，就要判断当前方法是否是 `options` ，如果是则直接返回，否则报出异常：

```
protected function getRouteForMethods($request, array $methods)
{
    if ($request->method() == 'OPTIONS') {
        return (new Route('OPTIONS', $request->path(), function
() use ($methods) {
            return new Response('', 200, ['Allow' => implode(','
, $methods)]);
        }))->bind($request);
    }

    $this->methodNotAllowed($methods);
}

protected function methodNotAllowed(array $others)
{
    throw new MethodNotAllowedHttpException($others);
}
```

前言

当进行了路由匹配与路由参数绑定后，接下来就要进行路由闭包或者控制器的运行，在此之前，本文先介绍中间件的相关源码。

中间件的搜集

由于定义的中间件方式很灵活，所以在运行控制器或者路由闭包之前，我们需要先将在各个地方注册的所有中间件都搜集到一起，然后集中排序。

```
public function dispatchToRoute(Request $request)
{
    $route = $this->findRoute($request);

    $request->setRouteResolver(function () use ($route) {
        return $route;
    });

    $this->events->dispatch(new Events\RouteMatched($route, $req
uest));

    $response = $this->runRouteWithinStack($route, $request);

    return $this->prepareResponse($request, $response);
}

protected function runRouteWithinStack(Route $route, Request $re
quest)
{
    $shouldSkipMiddleware = $this->container->bound('middleware.
disable') &&
                            $this->container->make('middleware.d
isable') === true;

    $middleware = $shouldSkipMiddleware ? [] : $this->gatherRout
eMiddleware($route);
```

```
    return (new Pipeline($this->container))
                    ->send($request)
                    ->through($middleware)
                    ->then(function ($request) use ($route) {
                        return $this->prepareResponse(
                            $request, $route->run()
                        );
                    });
}

public function gatherRouteMiddleware(Route $route)
{
    $middleware = collect($route->gatherMiddleware())->map(funct
ion ($name) {
        return (array) MiddlewareNameResolver::resolve($name, $t
his->middleware, $this->middlewareGroups);
    })->flatten();

    return $this->sortMiddleware($middleware);
}
```

路由的中间件大致有两个大的来源：

- 在路由的定义过程中，利用关键字 `middleware` 为路由添加中间件，这种中间件都是在文件 `App\Http\Kernel` 中 `$middlewareGroups` 、 `$routeMiddleware` 这两个数组定义的中间件别名。
- 在路由控制器的构造函数中，添加中间件，可以在这里定义一个闭包作为中间件，也可以利用中间件别名。

```
public function gatherMiddleware()
{
    if (! is_null($this->computedMiddleware)) {
        return $this->computedMiddleware;
    }

    $this->computedMiddleware = [];

    return $this->computedMiddleware = array_unique(array_merge(
        $this->middleware(), $this->controllerMiddleware()
    ), SORT_REGULAR);
}
```

路由定义的中间件是从 `action` 数组中取出来的：

```
 public function middleware($middleware = null)
{
    if (is_null($middleware)) {
        return (array) Arr::get($this->action, 'middleware', [])
;
    }

    if (is_string($middleware)) {
        $middleware = func_get_args();
    }

    $this->action['middleware'] = array_merge(
        (array) Arr::get($this->action, 'middleware', []), $midd
leware
    );

    return $this;
}
```

控制器定义的中间件：

```
public function controllerMiddleware()
{
    if (! $this->isControllerAction()) {
        return [];
    }

    return ControllerDispatcher::getMiddleware(
        $this->getController(), $this->getControllerMethod()
    );
}

public function getController()
{
    $class = $this->parseControllerCallback()[0];

    if (! $this->controller) {
        $this->controller = $this->container->make($class);
    }

    return $this->controller;
}

protected function getControllerMethod()
{
    return $this->parseControllerCallback()[1];
}

protected function parseControllerCallback()
{
    return Str::parseCallback($this->action['uses']);
}

public static function parseCallback($callback, $default = null)
{
    return static::contains($callback, '@') ? explode('@', $call
back, 2) : [$callback, $default];
}
```

当前的路由如果使用控制器的时候，就要解析属性 `use` ，解析出控制器的类名与类方法。接下来就需要 `ControllerDispatcher` 类。

在讲解 `ControllerDispatcher` 类之前，我们需要先了解一下控制器中间件：

```
abstract class Controller
{
    public function middleware($middleware, array $options = [])
    {
        foreach ((array) $middleware as $m) {
            $this->middleware[] = [
                'middleware' => $m,
                'options' => &$options,
            ];
        }

        return new ControllerMiddlewareOptions($options);
    }
}

class ControllerMiddlewareOptions
{
    protected $options;

    public function __construct(array &$options)
    {
        $this->options = &$options;
    }

    public function only($methods)
    {
        $this->options['only'] = is_array($methods) ? $methods :
 func_get_args();

        return $this;
    }

    public function except($methods)
    {
        $this->options['except'] = is_array($methods) ? $methods
```

```
 : func_get_args();

        return $this;
    }
}
```

在为控制器定义中间的是，可以为中间件利用 `only` 指定在当前控制器中调用该中间件的特定控制器方法，也可以利用 `except` 指定在当前控制器禁止调用中间件的方法。这些信息都保存在控制器的变量 `middleware` 的 `options` 中。

在搜集控制器的中间件时，就要利用中间件的这些信息：

```
class ControllerDispatcher
{
    public static function getMiddleware($controller, $method)
    {
        if (! method_exists($controller, 'getMiddleware')) {
            return [];
        }

        return collect($controller->getMiddleware())->reject(function ($data) use ($method) {
            return static::methodExcludedByOptions($method, $data['options']);
        })->pluck('middleware')->all();
    }

    protected static function methodExcludedByOptions($method, array $options)
    {
        return (isset($options['only']) && ! in_array($method, (array) $options['only'])) ||
            (! empty($options['except']) && in_array($method, (array) $options['except']));
    }
}
```

在 `ControllerDispatcher` 类中，利用了 `reject` 函数对每一个中间件都进行了控制器方法的判断，排除了不支持该控制器方法的中间件。`pluck` 函数获取了控制器 `$this->middleware[]` 数组中 `middleware` 的所有元素。

中间件的解析

中间件解析主要的工作是将路由中中间件的别名转化为中间件全程，主要流程为：

```
class MiddlewareNameResolver
{
    public static function resolve($name, $map, $middlewareGroup
s)
    {
        if ($name instanceof Closure) {
            return $name;
        } elseif (isset($map[$name]) && $map[$name] instanceof C
losure) {
            return $map[$name];

        } elseif (isset($middlewareGroups[$name])) {
            return static::parseMiddlewareGroup(
                $name, $map, $middlewareGroups
            );

        } else {
            list($name, $parameters) = array_pad(explode(':', $n
ame, 2), 2, null);

            return (isset($map[$name]) ? $map[$name] : $name).
                   (! is_null($parameters) ? ':'.$parameters : ''
);
        }
    }
}
```

可以看出，解析的中间件对象有三种：闭包、中间件别名、中间件组。

- 对于闭包来说，`resolve` 直接返回闭包；
- 对于中间件别名来说，例如 `auth` ，会从 `App\Http\Kernel` 文件 `$routeMiddleware` 数组中寻找中间件全名 `\Illuminate\Auth\Middleware\Authenticate::class`
- 对于具有参数的中间件别名来说，例如 `throttle:60,1` ,会将别名转化为全名 `\Illuminate\Routing\Middleware\ThrottleRequests::60,1`
- 对于中间件组来说，会调用 `parseMiddlewareGroup` 函数。

```
protected static function parseMiddlewareGroup($name, $map, $mid
dlewareGroups)
{
    $results = [];

    foreach ($middlewareGroups[$name] as $middleware) {
        if (isset($middlewareGroups[$middleware])) {
            $results = array_merge($results, static::parseMiddle
wareGroup(
                $middleware, $map, $middlewareGroups
            ));

            continue;
        }

        list($middleware, $parameters) = array_pad(
            explode(':', $middleware, 2), 2, null
        );

        if (isset($map[$middleware])) {
            $middleware = $map[$middleware];
        }

        $results[] = $middleware.($parameters ? ':'.$parameters
: '');
    }

    return $results;
}
```

可以看出，对于中间件组来说，就要从 `App\Http\Kernel` 文件 `$$middlewareGroups` 数组中寻找组内的多个中间件，例如中间件组 `api` ：

```
'api' => [
    'throttle:60,1',
    'bindings',
]
```

解析出的中间件可能存在参数，别名转化为全名后函数返回。值得注意的是，中间件组内不一定都是别名，也有可能是中间件组的组名，例如：

```
'api' => [
    'throttle:60,1',
    'web',
]

'web' => [
    \App\Http\Middleware\EncryptCookies::class,
    \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::cl
ass,
],
```

这时，就需要迭代解析。

中间件的排序

```
public function gatherRouteMiddleware(Route $route)
{
    $middleware = collect($route->gatherMiddleware())->map(funct
ion ($name) {
        return (array) MiddlewareNameResolver::resolve($name, $t
his->middleware, $this->middlewareGroups);
    })->flatten();

    return $this->sortMiddleware($middleware);
}
```

将所有中间件搜集并解析完毕后，接下来就要对中间件的调用顺序做一些调整，以确保中间件功能正常。

```
protected $middlewarePriority = [
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    \Illuminate\Auth\Middleware\Authenticate::class,
    \Illuminate\Session\Middleware\AuthenticateSession::class,
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
    \Illuminate\Auth\Middleware\Authorize::class,
];
```

数组 `middlewarePriority` 中保存着必须有一定顺序的中间件，例如 `StartSession` 中间件就必须运行在 `ShareErrorsFromSession` 之前。因此一旦路由中有这两个中间件，那么就要确保两者的顺序一致。

中间件的排序由函数 `sortMiddleware` 负责：

```
class SortedMiddleware extends Collection
{
    public function __construct(array $priorityMap, $middlewares)

    {
        if ($middlewares instanceof Collection) {
            $middlewares = $middlewares->all();
        }

        $this->items = $this->sortMiddleware($priorityMap, $midd
lewares);
    }

    protected function sortMiddleware($priorityMap, $middlewares)

    {
        $lastIndex = 0;

        foreach ($middlewares as $index => $middleware) {
            if (! is_string($middleware)) {
                continue;
```

```
            }

            $stripped = head(explode(':', $middleware));

            if (in_array($stripped, $priorityMap)) {
                $priorityIndex = array_search($stripped, $priori
tyMap);

                if (isset($lastPriorityIndex) && $priorityIndex
< $lastPriorityIndex) {
                    return $this->sortMiddleware(
                        $priorityMap, array_values(
                            $this->moveMiddleware($middlewares,
$index, $lastIndex)
                        )
                    );
                } else {
                    $lastIndex = $index;
                    $lastPriorityIndex = $priorityIndex;
                }
            }
        }

        return array_values(array_unique($middlewares, SORT_REGU
LAR));
    }

    protected function moveMiddleware($middlewares, $from, $to)
    {
        array_splice($middlewares, $to, 0, $middlewares[$from]);

        unset($middlewares[$from + 1]);

        return $middlewares;
    }
}
```

函数的方法很简单，检测当前中间件数组，查看是否存在中间件是数组 `middlewarePriority` 内元素。如果发现了两个中间件不符合顺序，那么就要调换中间件顺序，然后进行迭代。

前言

当路由与请求进行正则匹配后，各个路由的参数就获得了它们各自的数值。然而，有些路由参数变量，我们还想要把它转化为特定的对象，这时候就需要中间件的帮助。 `SubstituteBindings` 中间件就是一个将路由参数转化为特定对象的组件，它默认可以将特定名称的路由参数转化数据库模型对象，可以转化已绑定的路由参数为把绑定的对象。

`SubstituteBindings` 中间件的使用

数据库模型隐性转化

首先我们定义了一个带有路由参数的路由：

```
Route::put('user/{userid}', 'UserController@update');
```

然后我们在路由的控制器方法中或者路由闭包函数中定义一个数据库模型类型的参数，这个参数名与路由参数相同：

```
class UserController extends Controller
{
    public function update(UserModel $userid)
    {
        $userid->name = 'taylor';
        $userid->update();
    }
}
```

这时，路由的参数会被中间件隐性地转化为 `UserModel` ，且模型变量 `$userid` 的主键值为参数变量 `{userid}` 正则匹配后的数值。

综合测试样例：

```
public function testImplicitBindingsWithOptionalParameter()
```

```
{
    unset($_SERVER['__test.controller_callAction_parameters']);
    $router->get(($str = str_random()).'/{user}/{defaultNull?}/{
team?}', [
        'middleware' => SubstituteBindings::class,
        'uses' => 'Illuminate\Tests\Routing\RouteTestAnotherCont
rollerWithParameterStub@withModels',
    ]);

    $router->dispatch(Request::create($str.'/1', 'GET'));

    $values = array_values($_SERVER['__test.controller_callActio
n_parameters']);

    $this->assertEquals(1, $values[0]->value);
}


class RouteTestAnotherControllerWithParameterStub extends Contro
ller
{
    public function callAction($method, $parameters)
    {
        $_SERVER['__test.controller_callAction_parameters'] = $p
arameters;
    }

    public function withModels(RoutingTestUserModel $user)
    {
    }
}

class RoutingTestUserModel extends Model
{
    public function getRouteKeyName()
    {
        return 'id';
    }

    public function where($key, $value)
```

```
    {
        $this->value = $value;

        return $this;
    }

    public function first()
    {
        return $this;
    }

    public function firstOrFail()
    {
        return $this;
    }
}
```

路由显示绑定

除了隐示地转化路由参数外，我们还可以给路由参数显示提供绑定。显示绑定有 `bind` 、 `model` 两种方法。

- 通过 `bind` 为参数绑定闭包函数：

```
public function testRouteBinding()
{
    $router = $this->getRouter();
    $router->get('foo/{bar}', ['middleware' => SubstituteBindings::class, 'uses' => function ($name) {
         return $name;
    }]);
    $router->bind('bar', function ($value) {
        return strtoupper($value);
    });
    $this->assertEquals('TAYLOR', $router->dispatch(Request::create('foo/taylor', 'GET'))->getContent());
}
```

- 通过 `bind` 为参数绑定类方法，可以指定 `classname@method` ，也可以直接使用类名，默认会调用类的 `bind` 函数：

```
public function testRouteClassBinding()
{
    $router = $this->getRouter();
    $router->get('foo/{bar}', ['middleware' => SubstituteBinding
s::class, 'uses' => function ($name) {
        return $name;
    }]);
    $router->bind('bar', 'Illuminate\Tests\Routing\RouteBindingS
tub');
    $this->assertEquals('TAYLOR', $router->dispatch(Request::cre
ate('foo/taylor', 'GET'))->getContent());
}

public function testRouteClassMethodBinding()
{
    $router = $this->getRouter();
    $router->get('foo/{bar}', ['middleware' => SubstituteBinding
s::class, 'uses' => function ($name) {
        return $name;
    }]);
    $router->bind('bar', 'Illuminate\Tests\Routing\RouteBindingS
tub@find');
    $this->assertEquals('dragon', $router->dispatch(Request::cre
ate('foo/Dragon', 'GET'))->getContent());
}

class RouteBindingStub
{
    public function bind($value, $route)
    {
        return strtoupper($value);
    }

    public function find($value, $route)
    {
        return strtolower($value);
    }
}
```

- 通过 `model` 为参数绑定数据库模型，路由的参数就不需要和控制器方法中的变量名相同，`laravel` 会利用路由参数的值去调用 `where` 方法查找对应记录：

```
if ($model = $instance->where($instance->getRouteKeyName(), $val
ue)->first()) {
     return $model;
}
```

测试样例如下：

```
public function testModelBinding()
{
    $router = $this->getRouter();
    $router->get('foo/{bar}', ['middleware' => SubstituteBinding
s::class, 'uses' => function ($name) {
        return $name;
    }]);
    $router->model('bar', 'Illuminate\Tests\Routing\RouteModelBi
ndingStub');
    $this->assertEquals('TAYLOR', $router->dispatch(Request::cre
ate('foo/taylor', 'GET'))->getContent());
}

class RouteModelBindingStub
{
    public function getRouteKeyName()
    {
        return 'id';
    }

    public function where($key, $value)
    {
        $this->value = $value;

        return $this;
    }

    public function first()
    {
        return strtoupper($this->value);
    }
}
```

- 若绑定的 `model` 并没有找到对应路由参数的记录，可以在 `model` 中定义一个闭包函数，路由参数会调用闭包函数：

```
public function testModelBindingWithCustomNullReturn()
{
    $router = $this->getRouter();
```

```
    $router->get('foo/{bar}', ['middleware' => SubstituteBinding
s::class, 'uses' => function ($name) {
        return $name;
    }]);
    $router->model('bar', 'Illuminate\Tests\Routing\RouteModelBi
ndingNullStub', function () {
        return 'missing';
    });
    $this->assertEquals('missing', $router->dispatch(Request::cr
eate('foo/taylor', 'GET'))->getContent());
}

public function testModelBindingWithBindingClosure()
{
    $router = $this->getRouter();
    $router->get('foo/{bar}', ['middleware' => SubstituteBinding
s::class, 'uses' => function ($name) {
        return $name;
    }]);
    $router->model('bar', 'Illuminate\Tests\Routing\RouteModelBi
ndingNullStub', function ($value) {
        return (new RouteModelBindingClosureStub())->findAlterna
te($value);
    });
    $this->assertEquals('tayloralt', $router->dispatch(Request::
create('foo/TAYLOR', 'GET'))->getContent());
}

class RouteModelBindingNullStub
{
    public function getRouteKeyName()
    {
        return 'id';
    }

    public function where($key, $value)
    {
        return $this;
    }
```

```
    public function first()
    {
    }
}

class RouteModelBindingClosureStub
{
    public function findAlternate($value)
    {
        return strtolower($value).'alt';
    }
}
```

`SubstituteBindings` 中间件的源码解析

```
class SubstituteBindings
{
    public function handle($request, Closure $next)
    {
        $this->router->substituteBindings($route = $request->rou
te());

        $this->router->substituteImplicitBindings($route);

        return $next($request);
    }
}
```

从代码来看，`substituteBindings` 用于显示的参数转化，`substituteImplicitBindings` 用于隐性的参数转化。

隐性参数转化源码解析

进行隐性参数转化，其步骤为：

- 扫描控制器方法或者闭包函数所有的参数，提取出数据库模型类型对象

- 根据模型类型对象的 `name`，找出与模型对象命名相同的路由参数
- 根据模型类型对象的 `classname`，构建数据库模型类型对象，根据路由参数的数值在数据库中执行 `sql` 语句查询

```
public function substituteImplicitBindings($route)
{
    ImplicitRouteBinding::resolveForRoute($this->container, $rou
te);
}

class ImplicitRouteBinding
{
    public static function resolveForRoute($container, $route)
    {
        $parameters = $route->parameters();

        foreach ($route->signatureParameters(Model::class) as $p
arameter) {
            $class = $parameter->getClass();

            if (array_key_exists($parameter->name, $parameters)
&&
                ! $route->parameter($parameter->name) instanceof
 Model) {
                $method = $parameter->isDefaultValueAvailable()
? 'first' : 'firstOrFail';

                $model = $container->make($class->name);

                $route->setParameter(
                    $parameter->name, $model->where(
                        $model->getRouteKeyName(), $parameters[$
parameter->name]
                    )->{$method}()
                );
            }
        }
    }
}
```

值得注意的是，显示参数转化的优先级要高于隐性转化，如果当前参数已经被 `model` 函数显示转化，那么该参数并不会进行隐性转化，也就是上面语句 `!$route->parameter($parameter->name) instanceof Model` 的作用。

其中扫描控制器方法参数的功能主要利用反射机制：

```
public function signatureParameters($subClass = null)
{
    return RouteSignatureParameters::fromAction($this->action, $subClass);
}

class RouteSignatureParameters
{
    public static function fromAction(array $action, $subClass = null)
    {
        $parameters = is_string($action['uses'])
                        ? static::fromClassMethodString($action['uses'])
                        : (new ReflectionFunction($action['uses']))->getParameters();

        return is_null($subClass) ? $parameters : array_filter($parameters, function ($p) use ($subClass) {
            return $p->getClass() && $p->getClass()->isSubclassOf($subClass);
        });
    }

    protected static function fromClassMethodString($uses)
    {
        list($class, $method) = Str::parseCallback($uses);

        return (new ReflectionMethod($class, $method))->getParameters();
    }
}
```

bind 显示参数绑定

路由的 `bind` 功能由专门的 `binders` 数组负责，这个数组中保存着所有的需要显示转化的路由参数与他们的转化闭包函数：

```
public function bind($key, $binder)
{
    $this->binders[str_replace('-', '_', $key)] = RouteBinding::forCallback(
        $this->container, $binder
    );
}

class RouteBinding
{
    public static function forCallback($container, $binder)
    {
        if (is_string($binder)) {
            return static::createClassBinding($container, $binder);
        }

        return $binder;
    }

    protected static function createClassBinding($container, $binding)
    {
        return function ($value, $route) use ($container, $binding) {
            list($class, $method) = Str::parseCallback($binding, 'bind');

            $callable = [$container->make($class), $method];

            return call_user_func($callable, $value, $route);
        };
    }
}
```

可以看出，`bind` 函数可以绑定闭包、`classname@method`、`classname`，如果仅仅绑定了一个类名，那么程序默认调用类中 `bind` 方法。

model 显示参数绑定

`model` 调用 `bind` 函数，赋给 `bind` 函数一个提前包装好的闭包函数：

```
public function model($key, $class, Closure $callback = null)
{
    $this->bind($key, RouteBinding::forModel($this->container, $class, $callback));
}

class RouteBinding
{
    public static function forModel($container, $class, $callback = null)
    {
        return function ($value) use ($container, $class, $callback) {
            if (is_null($value)) {
                return;
            }

            $instance = $container->make($class);

            if ($model = $instance->where($instance->getRouteKeyName(), $value)->first()) {
                return $model;
            }

            if ($callback instanceof Closure) {
                return call_user_func($callback, $value);
            }

            throw (new ModelNotFoundException)->setModel($class);
        };
    }
}
```

可以看出，这个闭包函数与隐性转化很相似，都是首先创建数据库模型对象，再利用路由参数值来查询数据库，返回对象。 `model` 还可以提供默认的闭包函数，以供查询不到数据库时调用。

显示路由参数转化

当运行中间件 `SubstituteBindings` 时，就会将先前绑定的各个闭包函数执行，并对路由参数进行转化：

```
public function substituteBindings($route)
{
    foreach ($route->parameters() as $key => $value) {
        if (isset($this->binders[$key])) {
            $route->setParameter($key, $this->performBinding($ke
y, $value, $route));
        }
    }

    return $route;
}

protected function performBinding($key, $value, $route)
{
    return call_user_func($this->binders[$key], $value, $route);
}

public function setParameter($name, $value)
{
    $this->parameters();

    $this->parameters[$name] = $value;
}
```

前言

经过前面一系列中间件的工作，现在请求终于要达到了正确的控制器方法了。本篇文章主要讲 `laravel` 如何调用控制器方法，并且为控制器方法依赖注入构建参数的过程。

路由控制器的调用

我们前面已经解析过中间件的搜集与排序、pipeline 的原理，接下来就要进行路由的 `run` 运行函数：

```
protected function runRouteWithinStack(Route $route, Request $request)
{
    $shouldSkipMiddleware = $this->container->bound('middleware.disable') &&
                            $this->container->make('middleware.disable') === true;

    $middleware = $shouldSkipMiddleware ? [] : $this->gatherRouteMiddleware($route);

    return (new Pipeline($this->container))
                    ->send($request)
                    ->through($middleware)
                    ->then(function ($request) use ($route) {
                        return $this->prepareResponse(
                            $request, $route->run()
                        );
                    });
}
```

路由的 `run` 函数主要负责路由控制器方法与路由闭包函数的运行：

```php
public function run()
{
    $this->container = $this->container ?: new Container;

    try {
        if ($this->isControllerAction()) {
            return $this->runController();
        }

        return $this->runCallable();
    } catch (HttpResponseException $e) {
        return $e->getResponse();
    }
}
```

路由的运行主要靠 `ControllerDispatcher` 这个类：

```
class Route
{
    protected function isControllerAction()
    {
        return is_string($this->action['uses']);
    }

    protected function runController()
    {
        return (new ControllerDispatcher($this->container))->dis
patch(
            $this, $this->getController(), $this->getControllerM
ethod()
        );
    }
}

class ControllerDispatcher
{
    use RouteDependencyResolverTrait;

    public function dispatch(Route $route, $controller, $method)
    {
        $parameters = $this->resolveClassMethodDependencies(
            $route->parametersWithoutNulls(), $controller, $meth
od
        );

        if (method_exists($controller, 'callAction')) {
            return $controller->callAction($method, $parameters)
;
        }

        return $controller->{$method}(...array_values($parameter
s));
    }
}
```

上面可以很清晰地看出，控制器的运行分为两步：解析函数参数、调用callAction

解析控制器方法参数

解析参数的功能主要由 `ControllerDispatcher` 类的 `RouteDependencyResolverTrait` 这一 `trait` 负责：

```
trait RouteDependencyResolverTrait
{
    protected function resolveClassMethodDependencies(array $parameters, $instance, $method)
    {
        if (! method_exists($instance, $method)) {
            return $parameters;
        }

        return $this->resolveMethodDependencies(
            $parameters, new ReflectionMethod($instance, $method)
        );
    }

    public function resolveMethodDependencies(array $parameters, ReflectionFunctionAbstract $reflector)
    {
        $instanceCount = 0;

        $values = array_values($parameters);

        foreach ($reflector->getParameters() as $key => $parameter) {
            $instance = $this->transformDependency(
                $parameter, $parameters
            );

            if (! is_null($instance)) {
                $instanceCount++;

                $this->spliceIntoParameters($parameters, $key, $instance);
            } elseif (! isset($values[$key - $instanceCount]) &&
```

```
                        $parameter->isDefaultValueAvailable()) {
                  $this->spliceIntoParameters($parameters, $key, $
parameter->getDefaultValue());
            }
        }

        return $parameters;
    }
}
```

控制器方法函数参数构造难点在于，参数来源有三种：

- 路由参数赋值
- Ioc 容器自动注入
- 函数自带默认值

在 Ioc 容器自动注入的时候，要保证路由的现有参数中没有相应的类，防止依赖注入覆盖路由绑定的参数：

```
protected function transformDependency(ReflectionParameter $para
meter, $parameters)
{
    $class = $parameter->getClass();

    if ($class && ! $this->alreadyInParameters($class->name, $pa
rameters)) {
        return $this->container->make($class->name);
    }
}

protected function alreadyInParameters($class, array $parameters)

{
    return ! is_null(Arr::first($parameters, function ($value) u
se ($class) {
        return $value instanceof $class;
    }));
}
```

由 Ioc 容器构造出的参数需要插入到原有的路由参数数组中：

```
if (! is_null($instance)) {
    $instanceCount++;

    $this->spliceIntoParameters($parameters, $key, $instance);
}

protected function spliceIntoParameters(array &$parameters, $off
set, $value)
{
    array_splice(
        $parameters, $offset, 0, [$value]
    );
}
```

当路由的参数数组与 Ioc 容器构造的参数数量不足以覆盖控制器参数个数时，就要去判断控制器是否具有默认参数：

```
elseif (! isset($values[$key - $instanceCount]) &&
       $parameter->isDefaultValueAvailable()) {
    $this->spliceIntoParameters($parameters, $key, $parameter->g
etDefaultValue());
}
```

调用控制器方法 callAction

所有的控制器并非是直接调用相应方法的，而是通过 `callAction` 函数再分配，如果实在没有相应方法还会调用魔术方法 `__call()`：

```
public function callAction($method, $parameters)
{
    return call_user_func_array([$this, $method], $parameters);
}

public function __call($method, $parameters)
{
    throw new BadMethodCallException("Method [{$method}] does no
t exist.");
}
```

路由闭包函数的调用

路由闭包函数的调用与控制器方法一样，仍然需要依赖注入，参数构造：

```
protected function runCallable()
{
    $callable = $this->action['uses'];

    return $callable(...array_values($this->resolveMethodDepende
ncies(
        $this->parametersWithoutNulls(), new ReflectionFunction(
$this->action['uses'])
    )));
}
```

前言

我们在前面的文章已经讲了整个路由与控制器的源码，我们今天这个文章开始向大家介绍在 `laravel` 中创建 `RESTFul` 风格的控制器。

关于什么是RESTFul风格及其规范可参考这篇文章：理解RESTful架构。

关于 `laravel` 中 `RESTFul` 风格控制器的创建简要介绍 ：HTTP控制器实例教程 —— 创建 `RESTFul` 风格控制器实现文章增删改查

创建 `RESTFul` 风格控制器

要想在 `laravel` 中创建 `RESTFul` 风格控制器，只需要一句：

```
Route::resource('post','PostController');
```

该路由包含了指向多个动作的子路由：

方法	路径	动作	路由名称
GET	/post	index	post.index
GET	/post/create	create	post.create
POST	/post	store	post.store
GET	/post/{post}	show	post.show
GET	/post/{post}/edit	edit	post.edit
PUT/PATCH	/post/{post}	update	post.update
DELETE	/post/{post}	destroy	post.destroy

这种用法既简单又方便，接下来，我们将会说一下 `laravel` 为我们提供的更加灵活的用法。

前缀 `RESTFul` 路由

可以为 `RESTFul` 路由定义前缀：

```
$router->resource('prefix/foos', 'FooController');

$this->assertEquals('prefix/foos/{foo}', $routes[3]->uri());
```

双参数 `RESTFul` 路由

`laravel` 允许定义拥有两个参数的 `RESTFul` 路由：

```
$router->resource('foos.bars', 'FooController');

$this->assertEquals('foos/{foo}/bars/{bar}', $routes[3]->uri());
```

参数自定义命名

一般来说，`RESTFul` 路由的参数命名规则是路由单数，符号 `-` 转为 `_`，例如下面例子中 `bars`，和 `foo-baz`。

```
$router->resource('foos', 'FooController');
$this->assertEquals('foos/{foo}', $routes[3]->uri());

$router->resource('foo-bar.foo-baz', 'FooController', ['only' =>
 ['show']]);
$this->assertEquals('foo-bar/{foo_bar}/foo-baz/{foo_baz}', $rout
es[0]->uri());
```

我们可以利用 `parameters` 强制这种单数模式：

```
$router->resource('foos', 'FooController', ['parameters' => 'sin
gular']);
$this->assertEquals('foos/{foo}', $routes[3]->uri());
```

我们也可以利用 `singularParameters` 来强制：

```
ResourceRegistrar::singularParameters(true);

$router->resource('foos', 'FooController', ['parameters' => 'singular']);
$this->assertEquals('foos/{foo}', $routes[3]->uri());
```

我们还可以不使用单数，利用 `parameters` 用自己自定义的名字来定义参数：

```
$router->resource('bars.foos.bazs', 'FooController', ['parameters' => ['foos' => 'oof', 'bazs' => 'b']]);

$this->assertEquals('bars/{bar}/foos/{oof}/bazs/{b}', $routes[3]->uri());
```

同时，我们仍然可以利用 `setParameters` 函数来自定义参数命名：

```
ResourceRegistrar::setParameters(['foos' => 'oof', 'bazs' => 'b']);

$router->resource('bars.foos.bazs', 'FooController');
$this->assertEquals('bars/{bar}/foos/{oof}/bazs/{b}', $routes[3]->uri());
```

`RESTFul` 路由动词控制

`laravel` 为 `RESTFul` 路由生成了两个带有动词的路由：`create` 、`edit` ，分别用于加载订单的创建页面与编辑页面，这两个动词 `laravel` 是允许修改的：

```
ResourceRegistrar::verbs([
    'create' => 'ajouter',
    'edit' => 'modifier',
]);

$router->resource('foo', 'FooController');
$routes = $router->getRoutes();

$this->assertEquals('foo/ajouter', $routes->getByName('foo.create')->uri());
$this->assertEquals('foo/{foo}/modifier', $routes->getByName('foo.edit')->uri());
```

控制器方法约束

一般情况下，我们都会一次性想要上面所生成的七个路由，然而，有时候，我们只需要其中几个，或者不想要其中几个。这时候就可以利用 `only` 或者 `except` :

```
$router = $this->getRouter();
$router->resource('foo', 'FooController', ['only' => ['show', 'destroy']]);
$routes = $router->getRoutes();

$this->assertCount(2, $routes);
```

```
$router = $this->getRouter();
$router->resource('foo', 'FooController', ['except' => ['show', 'destroy']]);
$routes = $router->getRoutes();

$this->assertCount(5, $routes);
```

`RESTFul` 路由名称自定义

`RESTFul` 路由的每个路由都要自己默认的路由名称， `laravel` 允许我们对路由名称进行修改：

我们可以用 `as` 来为路由名称添加前缀：

```
$router->resource('foo-bars', 'FooController', ['only' => ['show'
], 'as' => 'prefix']);

$this->assertEquals('prefix.foo-bars.show', $routes[0]->getName(
));
```

当有多个路由参数的时候，路由参数默认添加到了路由名称中：

```
$router->resource('prefix/foo.bar', 'FooController');

$this->assertTrue($router->getRoutes()->hasNamedRoute('foo.bar.i
ndex'));
```

可以利用 `names` 为单个路由来命名：

```
$router->resource('foo', 'FooController', ['names' => [
    'index' => 'foo',
    'show' => 'bar',
]]);

$this->assertTrue($router->getRoutes()->hasNamedRoute('foo'));
$this->assertTrue($router->getRoutes()->hasNamedRoute('bar'));
```

还可以利用 `names` 为所有路由来命名：

```
$router->resource('foo', 'FooController', ['names' => 'bar']);

$this->assertTrue($router->getRoutes()->hasNamedRoute('bar.index'
));
```

RESTFul 路由源码分析

`RESTFul` 路由的创建工作由类 `ResourceRegistrar` 负责，这个类为默认为用户创建七个路由，函数方法 `register` 是创建路由的主函数：

```
class ResourceRegistrar
{
    public function register($name, $controller, array $options
= [])
    {
        if (isset($options['parameters']) && ! isset($this->para
meters)) {
            $this->parameters = $options['parameters'];
        }

        if (Str::contains($name, '/')) {
            $this->prefixedResource($name, $controller, $options
);

            return;
        }

        $base = $this->getResourceWildcard(last(explode('.', $na
me)));

        $defaults = $this->resourceDefaults;

        foreach ($this->getResourceMethods($defaults, $options)
as $m) {
            $this->{'addResource'.ucfirst($m)}($name, $base, $co
ntroller, $options);
        }
    }
}
```

这个函数主要流程分为三段：

- 判断是否由前缀
- 获取路由的基础参数

- 添加路由

拥有前缀的 `RESTFul` 路由

如果我们为 `RESTFul` 路由添加了前缀，那么 `laravel` 将会以 `group` 的形式添加路由：

```
protected function prefixedResource($name, $controller, array $options)
{
    list($name, $prefix) = $this->getResourcePrefix($name);

    $callback = function ($me) use ($name, $controller, $options) {
        $me->resource($name, $controller, $options);
    };

    return $this->router->group(compact('prefix'), $callback);
}

protected function getResourcePrefix($name)
{
    $segments = explode('/', $name);

    $prefix = implode('/', array_slice($segments, 0, -1));

    return [end($segments), $prefix];
}
```

获取基础 `RESTFul` 路由参数

在添加各种路由之前，我们需要先获取路由的基础参数，也就是当存在多参数情况下，最后的参数。获取参数后，如果用户有自定义命名，则获取自定义命名：

```
public function getResourceWildcard($value)
{
    if (isset($this->parameters[$value])) {
        $value = $this->parameters[$value];
    } elseif (isset(static::$parameterMap[$value])) {
        $value = static::$parameterMap[$value];
    } elseif ($this->parameters === 'singular' || static::$singu
larParameters) {
        $value = Str::singular($value);
    }

    return str_replace('-', '_', $value);
}
```

添加各种路由

添加路由主要有三个步骤：

- 计算路由 `uri`
- 获取路由属性
- 创建路由

```
protected function addResourceIndex($name, $base, $controller, $
options)
{
    $uri = $this->getResourceUri($name);

    $action = $this->getResourceAction($name, $controller, 'inde
x', $options);

    return $this->router->get($uri, $action);
}
```

当计算路由 `uri` 时，由于存在多参数的情况，需要循环计算路由参数：

```
public function getResourceUri($resource)
{
    if (! Str::contains($resource, '.')) {
        return $resource;
    }

    $segments = explode('.', $resource);

    $uri = $this->getNestedResourceUri($segments);

    return str_replace('/{'.$this->getResourceWildcard(end($segm
ents)).'}', '', $uri);
}

protected function getNestedResourceUri(array $segments)
{
    return implode('/', array_map(function ($s) {
        return $s.'/{'.$this->getResourceWildcard($s).'}';
    }, $segments));
}
```

当计算路由的属性时，最重要的是获取路由的名字，路由的名字可以是默认，也可以是用户利用 `names` 或者 `as` 属性来自定义：

```
protected function getResourceAction($resource, $controller, $method, $options)
{
    $name = $this->getResourceRouteName($resource, $method, $options);

    $action = ['as' => $name, 'uses' => $controller.'@'.$method];

    if (isset($options['middleware'])) {
        $action['middleware'] = $options['middleware'];
    }

    return $action;
}

protected function getResourceRouteName($resource, $method, $options)
{
    $name = $resource;

    if (isset($options['names'])) {
        if (is_string($options['names'])) {
            $name = $options['names'];
        } elseif (isset($options['names'][$method])) {
            return $options['names'][$method];
        }
    }

    $prefix = isset($options['as']) ? $options['as'].'.' : '';

    return trim(sprintf('%s%s.%s', $prefix, $name, $method), '.');
}
```

值得注意的是，如果单独为某一个方法命名，那么直接回返回命名，而不会受 `as` 和方法名 'method' 的影响。

前言

`laravel` 为我们提供便携的重定向功能，可以由门面 `Redirect` ，或者全局函数 `redirect()` 来启用，本篇文章将会介绍重定向功能的具体细节及源码分析。

URI 重定向

重定向功能是由类 `UrlGenerator` 所实现，这个类需要 `request` 来进行初始化：

```
 $url = new UrlGenerator(
    $routes = new RouteCollection,
    $request = Request::create('http://www.foo.com/')
);
```

重定向到 uri

- 当我们想要重定向到某个地址时，可以使用 `to` 函数：

```
$this->assertEquals('http://www.foo.com/foo/bar', $url->to('foo/bar'));
```

- 当我们想要添加额外的路径，可以将数组赋给第二个参数：

```
$this->assertEquals('https://www.foo.com/foo/bar/baz/boom', $url->to('foo/bar', ['baz', 'boom'], true));
$this->assertEquals('https://www.foo.com/foo/bar/baz?foo=bar', $url->to('foo/bar?foo=bar', ['baz'], true));
```

强制 https

如果我们想要重定向到 `https` ，我们可以设置第三个参数为 `true` ：

```
$this->assertEquals('https://www.foo.com/foo/bar', $url->to('foo/bar', [], true));
```

或者使用 `forceScheme` 函数：

```
$url->forceScheme('https');

$this->assertEquals('https://www.foo.com/foo/bar', $url->to('foo/bar'));
```

强制域名

```
$url->forceRootUrl('https://www.bar.com');

$this->assertEquals('https://www.bar.com/foo/bar', $url->to('foo/bar'));
```

路径自定义

```
$url->formatPathUsing(function ($path) {
    return '/something'.$path;
});

$this->assertEquals('http://www.foo.com/something/foo/bar', $url->to('foo/bar'));
```

路由重定向

重定向另一个非常重要的功能是重定向到路由所在的地址中去：

```
$route = new Route(['GET'], '/named-route', ['as' => 'plain']);
$routes->add($route);

$this->assertEquals('http:/www.bar.com/named-route', $url->route(
'plain'));
```

非域名路径

`laravel` 路由重定向可以选择重定向后的地址是否仍然带有域名，这个特性由第三个参数决定：

```
$route = new Route(['GET'], '/named-route', ['as' => 'plain']);
$routes->add($route);

$this->assertEquals('/named-route', $url->route('plain', [], fal
se));
```

重定向端口号

路由重定向可以允许带有 `request` 自己的端口：

```
$url = new UrlGenerator(
    $routes = new RouteCollection,
    $request = Request::create('http://www.foo.com:8080/')
);

$route = new Route(['GET'], 'foo/bar/{baz}', ['as' => 'bar', 'do
main' => 'sub.{foo}.com']);
$routes->add($route);

$this->assertEquals('http://sub.taylor.com:8080/foo/bar/otwell',
 $url->route('bar', ['taylor', 'otwell']));
```

重定向路径参数绑定

如果路由中含有参数，可以将需要的参数赋给 `route` 第二个参数：

```
$route = new Route(['GET'], 'foo/bar/{baz}', ['as' => 'foobar'])
;
$routes->add($route);

$this->assertEquals('http://www.foo.com/foo/bar/taylor', $url->r
oute('foobar', 'taylor'));
```

也可以根据参数的命名来指定参数绑定：

```
$route = new Route(['GET'], 'foo/bar/{baz}/breeze/{boom}', ['as'
 => 'bar']);
$routes->add($route);

$this->assertEquals('http://www.foo.com/foo/bar/otwell/breeze/ta
ylor', $url->route('bar', ['boom' => 'taylor', 'baz' => 'otwell'
]));
```

还可以利用 `defaults` 函数为重定向提供默认的参数来绑定：

```
$url->defaults(['locale' => 'en']);
$route = new Route(['GET'], 'foo', ['as' => 'defaults', 'domain'
 => '{locale}.example.com', function () {
}]);
$routes->add($route);

$this->assertEquals('http://en.example.com/foo', $url->route('de
faults'));
```

重定向路由 querystring 添加

当在 `route` 函数中赋给的参数多于路径参数的时候，多余的参数会被添加到 `querystring` 中：

```
$route = new Route(['GET'], 'foo/bar/{baz}/breeze/{boom}', ['as'
 => 'bar']);
$routes->add($route);

$this->assertEquals('http://www.foo.com/foo/bar/taylor/breeze/ot
well?fly=wall', $url->route('bar', ['taylor', 'otwell', 'fly' =>
'wall']));
```

fragment 重定向

```
$route = new Route(['GET'], 'foo/bar#derp', ['as' => 'fragment']
);
$routes->add($route);

$this->assertEquals('/foo/bar?baz=%C3%A5%CE%B1%D1%84#derp', $url
->route('fragment', ['baz' => 'åαф'], false));
```

路由 action 重定向

我们不仅可以通过路由的别名来重定向，还可以利用路由的控制器方法来重定向：

```
$route = new Route(['GET'], 'foo/bam', ['controller' => 'foo@bar'
]);
$routes->add($route);

$this->assertEquals('http://www.foo.com/foo/bam', $url->action('
foo@bar'));
```

可以设定重定向控制器的默认命名空间：

```
$url->setRootControllerNamespace('namespace');

$route = new Route(['GET'], 'foo/bar', ['controller' => 'namespa
ce\foo@bar']);
        $routes->add($route);

$route = new Route(['GET'], 'something/else', ['controller' => '
something\foo@bar']);
$routes->add($route);

$this->assertEquals('http://www.foo.com/foo/bar', $url->action('
foo@bar'));
$this->assertEquals('http://www.foo.com/something/else', $url->a
ction('\something\foo@bar'));
```

UrlRoutable 参数绑定

可以为重定向传入 `UrlRoutable` 类型的参数，重定向会通过类方法 `getRouteKey` 来获取对象的某个属性，进而绑定到路由的参数中去。

```
public function testRoutableInterfaceRoutingWithSingleParameter()

{
    $url = new UrlGenerator(
        $routes = new RouteCollection,
        $request = Request::create('http://www.foo.com/')
    );

    $route = new Route(['GET'], 'foo/{bar}', ['as' => 'routable'
]);
    $routes->add($route);

    $model = new RoutableInterfaceStub;
    $model->key = 'routable';

    $this->assertEquals('/foo/routable', $url->route('routable',
 $model, false));
}

class RoutableInterfaceStub implements UrlRoutable
{
    public $key;

    public function getRouteKey()
    {
        return $this->{$this->getRouteKeyName()};
    }

    public function getRouteKeyName()
    {
        return 'key';
    }
}
```

URI 重定向源码分析

在说重定向的源码之前，我们先了解一下一般的 `uri` 基本组成：

```
scheme://domain:port/path?queryString
```

也就是说，一般 `uri` 由五部分构成。重定向实际上就是按照各种传入的参数以及属性的设置来重新生成上面的五部分：

```
public function to($path, $extra = [], $secure = null)
{
    if ($this->isValidUrl($path)) {
        return $path;
    }

    $tail = implode('/', array_map(
        'rawurlencode', (array) $this->formatParameters($extra))
    );

    $root = $this->formatRoot($this->formatScheme($secure));

    list($path, $query) = $this->extractQueryString($path);

    return $this->format(
        $root, '/'.trim($path.'/'.$tail, '/')
    ).$query;
}
```

重定向 scheme

重定向的 `scheme` 由函数 `formatScheme` 生成：

```
public function formatScheme($secure)
{
    if (! is_null($secure)) {
        return $secure ? 'https://' : 'http://';
    }

    if (is_null($this->cachedSchema)) {
        $this->cachedSchema = $this->forceScheme ?: $this->reque
st->getScheme().'://';
    }

    return $this->cachedSchema;
}

public function forceScheme($schema)
{
    $this->cachedSchema = null;

    $this->forceScheme = $schema.'://';
}
```

可以看出来， `scheme` 的生成存在优先级：

- 由 `to` 传入的 `secure` 参数
- 由 `forceScheme` 设置的 `schema` 参数
- `request` 自带的 `scheme`

重定向 domain

重定向的 `domain` 由函数 `formatRoot` 生成：

```
public function formatRoot($scheme, $root = null)
{
    if (is_null($root)) {
        if (is_null($this->cachedRoot)) {
            $this->cachedRoot = $this->forcedRoot ?: $this->requ
est->root();
        }

        $root = $this->cachedRoot;
    }

    $start = Str::startsWith($root, 'http://') ? 'http://' : 'ht
tps://';

    return preg_replace('~'.$start.'~', $scheme, $root, 1);
}

public function forceRootUrl($root)
{
    $this->forcedRoot = rtrim($root, '/');

    $this->cachedRoot = null;
}
```

与 `scheme` 类似，`root` 的生成也存在优先级：

- 由 `to` 传入的 `root` 参数
- 由 `forceRootUrl` 设置的 `root` 参数
- `request` 自带的 `root`

重定向 path

重定向的 `path` 由三部分构成，一部分是 `request` 自带的 `path`，一部分是函数 `to` 原有的 `path`，另一部分是函数 `to` 传入的参数：

```
public function formatParameters($parameters)
{
    $parameters = array_wrap($parameters);

    foreach ($parameters as $key => $parameter) {
        if ($parameter instanceof UrlRoutable) {
            $parameters[$key] = $parameter->getRouteKey();
        }
    }

    return $parameters;
}

protected function extractQueryString($path)
{
    if (($queryPosition = strpos($path, '?')) !== false) {
        return [
            substr($path, 0, $queryPosition),
            substr($path, $queryPosition),
        ];
    }

    return [$path, ''];
}
```

路由重定向源码分析

相对于 `uri` 的重定向来说，路由重定向的 `scheme` 、 `root` 、 `path` 、 `queryString` 都要以路由自身的属性为第一优先级，此外还要利用额外参数来绑定路由的 `uri` 参数：

```
 public function route($name, $parameters = [], $absolute = true)

{
    if (! is_null($route = $this->routes->getByName($name))) {
        return $this->toRoute($route, $parameters, $absolute);
    }

    throw new InvalidArgumentException("Route [{$name}] not defi
ned.");
}

public function to($route, $parameters = [], $absolute = false)
{
    $domain = $this->getRouteDomain($route, $parameters);

    $uri = $this->addQueryString($this->url->format(
        $root = $this->replaceRootParameters($route, $domain, $p
arameters),
        $this->replaceRouteParameters($route->uri(), $parameters
)
    ), $parameters);

    if (preg_match('/\{.*?\}/', $uri)) {
        throw UrlGenerationException::forMissingParameters($rout
e);
    }

    $uri = strtr(rawurlencode($uri), $this->dontEncode);

    if (! $absolute) {
        return '/'.ltrim(str_replace($root, '', $uri), '/');
    }

    return $uri;
}
```

路由重定向 scheme

路由的重定向 `scheme` 需要先判断路由的 `scheme` 属性：

```
protected function getRouteScheme($route)
{
    if ($route->httpOnly()) {
        return 'http://';
    } elseif ($route->httpsOnly()) {
        return 'https://';
    } else {
        return $this->url->formatScheme(null);
    }
}
```

路由重定向 domain

```

public function to($route, $parameters = [], $absolute = false)
{
    $domain = $this->getRouteDomain($route, $parameters);

    $uri = $this->addQueryString($this->url->format(
        $root = $this->replaceRootParameters($route, $domain, $p
arameters),
        $this->replaceRouteParameters($route->uri(), $parameters
)
    ), $parameters);
    ...
}

protected function getRouteDomain($route, &$parameters)
{
    return $route->domain() ? $this->formatDomain($route, $param
eters) : null;
}

protected function formatDomain($route, &$parameters)
{
    return $this->addPortToDomain(
```

```
            $this->getRouteScheme($route).$route->domain()
        );
    }

    protected function addPortToDomain($domain)
    {
        $secure = $this->request->isSecure();

        $port = (int) $this->request->getPort();

        return ($secure && $port === 443) || (! $secure && $port ===
80)
                    ? $domain : $domain.':'.$port;
    }

    protected function replaceRootParameters($route, $domain, &$para
meters)
    {
        $scheme = $this->getRouteScheme($route);

        return $this->replaceRouteParameters(
            $this->url->formatRoot($scheme, $domain), $parameters
        );
    }
```

可以看出路由重定向时，域名的生成主要先经过函数 `getRouteDomain`，判断路由是否有 `domain` 属性,如果有域名属性，则将会作为 `formatRoot` 函数的参数传入，否则就会默认启动 1 `uri` 重定向的域名生成方法。

路由重定向参数绑定

路由重定向可以利用函数 `replaceRootParameters` 在域名当中参数绑定，，也可以在路径当中利用函数 `replaceRouteParameters` 进行参数绑定。参数绑定分为命名参数绑定与匿名参数绑定：

```
protected function replaceRouteParameters($path, array &$paramet
ers)
{
    $path = $this->replaceNamedParameters($path, $parameters);

    $path = preg_replace_callback('/\{.*?\}/', function ($match)
use (&$parameters) {
        return (empty($parameters) && ! Str::endsWith($match[0],
'?}'))
                    ? $match[0]
                    : array_shift($parameters);
    }, $path);

    return trim(preg_replace('/\{.*?\?\}/', '', $path), '/');
}
```

对于命名参数绑定,程序会分别从变量列表、默认变量列表中获取并替换路由参数对应的数值，若不存在该参数，则直接返回：

```
protected function replaceNamedParameters($path, &$parameters)
{
    return preg_replace_callback('/\{(.*?)\??\}/', function ($m)
use (&$parameters) {
        if (isset($parameters[$m[1]])) {
            return Arr::pull($parameters, $m[1]);
        } elseif (isset($this->defaultParameters[$m[1]])) {
            return $this->defaultParameters[$m[1]];
        } else {
            return $m[0];
        }
    }, $path);
}
```

命名参数绑定结束后，剩下的未被替换的路由参数将会被未命名的变量按顺序来替换。

路由重定向 queryString

如果变量列表在绑定路由后仍然有剩余，那么变量将会作为路由的 `queryString` ：

```
protected function addQueryString($uri, array $parameters)
{
    if (! is_null($fragment = parse_url($uri, PHP_URL_FRAGMENT))
) {
        $uri = preg_replace('/#.*/', '', $uri);
    }

    $uri .= $this->getRouteQueryString($parameters);

    return is_null($fragment) ? $uri : $uri."#{$fragment}";
}

protected function getRouteQueryString(array $parameters)
{
    if (count($parameters) == 0) {
        return '';
    }

    $query = http_build_query(
        $keyed = $this->getStringParameters($parameters)
    );


    if (count($keyed) < count($parameters)) {
        $query .= '&'.implode(
            '&', $this->getNumericParameters($parameters)
        );
    }

    return '?'.trim($query, '&');
}
```

路由重定向结束

路由 `uri` 构建完成后，将会继续判断是否存在违背绑定的路由参数，是否显示 `absolute` 的路由地址

```
public function to($route, $parameters = [], $absolute = false)
{
    ...
    if (preg_match('/\{.*?\}/', $uri)) {
        throw UrlGenerationException::forMissingParameters($rout
e);
    }

    $uri = strtr(rawurlencode($uri), $this->dontEncode);

    if (! $absolute) {
        return '/'.ltrim(str_replace($root, '', $uri), '/');
    }

    return $uri;
}
```

前言

`laravel` 在启动时，会加载项目的 `env` 文件，本文将会详细介绍 `env` 文件的使用与源码的分析。

ENV 文件的使用

多环境 ENV 文件的设置

一、在项目写多个 `ENV` 文件，例如三个 `env` 文件：

- `.env.development` 、
- `.env.staging` 、
- `.env.production` ，

这三个文件中分别针对不同环境为某些变量配置了不同的值，

二、配置 `APP_ENV` 环境变量值

配置环境变量的方法有很多，其中一个方法是在 `nginx` 的配置文件中写下这句代码：

```
fastcgi_param  APP_ENV  production;
```

那么 `laravel` 会通过 `env('APP_ENV')` 根据环境变量 `APP_ENV` 来判断当前具体的环境，假如环境变量 `APP_ENV` 为 `production` ，那么 `laravel` 将会自动加载 `.env.production` 文件。

自定义 ENV 文件的路径与文件名

`laravel` 为用户提供了自定义 `ENV` 文件路径或文件名的函数，

例如，若想要自定义 `env` 路径，就可以在 `bootstrap` 文件夹中 `app.php` 文件：

```
$app = new Illuminate\Foundation\Application(
    realpath(__DIR__.'/../')
);

$app->useEnvironmentPath('/customer/path')
```

若想要自定义 `env` 文件名称，就可以在 `bootstrap` 文件夹中 `app.php` 文件：

```
$app = new Illuminate\Foundation\Application(
    realpath(__DIR__.'/../')
);

$app->loadEnvironmentFrom('customer.env')
```

ENV 文件变量设置

- 在 `env` 文件中，我们可以为变量赋予具体值：

```
CFOO=bar
```

值得注意的是，这种具体值不允许赋予多个，例如：

```
CFOO=bar baz
```

- 可以为变量赋予字符串引用

```
CQUOTES="a value with a # character"
```

值得注意的是，这种引用不允许字符串中存在符号 `\` ，只能使用转义字符 `\\` 而且也不允许内嵌符号 `""` ，只能使用转移字符 `\"` ,否则取值会意外结束：

```
CQUOTESWITHQUOTE="a value with a # character & a quote \" charac
ter inside quotes" # " this is a comment

$this->assertEquals('a value with a # character & a quote " char
acter inside quotes', getenv('CQUOTESWITHQUOTE'));
```

- 可以在 `env` 文件中添加注释，方法是以 `#` 开始：

```
CQUOTES="a value with a # character" # this is a comment
```

- 可以使用 `export` 来为变量赋值：

```
export EFOO="bar"
```

- 可以在 `env` 文件中使用变量为变量赋值：

```
NVAR1="Hello"
NVAR2="World!"
NVAR3="{$NVAR1} {$NVAR2}"
NVAR4="${NVAR1} ${NVAR2}"
NVAR5="$NVAR1 {NVAR2}"

$this->assertEquals('{$NVAR1} {$NVAR2}', $_ENV['NVAR3']); // not
 resolved
$this->assertEquals('Hello World!', $_ENV['NVAR4']);
$this->assertEquals('$NVAR1 {NVAR2}', $_ENV['NVAR5']); // not re
solved
```

ENV 加载源码分析

laravel 加载 ENV

`ENV` 的加载功能由类 `\Illuminate\Foundation\Bootstrap\LoadEnvironmentVariables::class` 完成，它的启动函数为：

```
public function bootstrap(Application $app)
{
    if ($app->configurationIsCached()) {
        return;
    }

    $this->checkForSpecificEnvironmentFile($app);

    try {
        (new Dotenv($app->environmentPath(), $app->environmentFi
le()))->load();
    } catch (InvalidPathException $e) {
        //
    }
}
```

如果我们在环境变量中设置了 `APP_ENV` 变量，那么就会调用函数 `checkForSpecificEnvironmentFile` 来根据环境加载不同的 `env` 文件：

```
 protected function checkForSpecificEnvironmentFile($app)
{
    if (php_sapi_name() == 'cli' && with($input = new ArgvInput)
->hasParameterOption('--env')) {
        $this->setEnvironmentFilePath(
            $app, $app->environmentFile().'.'.$input->getParamet
erOption('--env')
        );
    }

    if (! env('APP_ENV')) {
        return;
    }

    $this->setEnvironmentFilePath(
        $app, $app->environmentFile().'.'.env('APP_ENV')
    );
}

protected function setEnvironmentFilePath($app, $file)
{
    if (file_exists($app->environmentPath().'/'.$file)) {
        $app->loadEnvironmentFrom($file);
    }
}
```

vlucas/phpdotenv 源码解读

`laravel` 中对 `env` 文件的读取是采用 `vlucas/phpdotenv` 的开源项目：

```
class Dotenv
{
    public function __construct($path, $file = '.env')
    {
        $this->filePath = $this->getFilePath($path, $file);
        $this->loader = new Loader($this->filePath, true);
    }

    public function load()
    {
        return $this->loadData();
    }

    protected function loadData($overload = false)
    {
        $this->loader = new Loader($this->filePath, !$overload);

        return $this->loader->load();
    }
}
```

`env` 文件变量的读取依赖类 `/Dotenv/Loader` :

```
class Loader
{
    public function load()
    {
        $this->ensureFileIsReadable();

        $filePath = $this->filePath;
        $lines = $this->readLinesFromFile($filePath);
        foreach ($lines as $line) {
            if (!$this->isComment($line) && $this->looksLikeSett
er($line)) {
                $this->setEnvironmentVariable($line);
            }
        }

        return $lines;
    }
}
```

我们可以看到， `env` 文件的读取的流程：

- 判断 `env` 文件是否可读
- 读取整个 `env` 文件，并将文件按行存储
- 循环读取每一行，略过注释
- 进行环境变量赋值

```
protected function ensureFileIsReadable()
{
    if (!is_readable($this->filePath) || !is_file($this->filePat
h)) {
        throw new InvalidPathException(sprintf('Unable to read t
he environment file at %s.', $this->filePath));
    }
}

protected function readLinesFromFile($filePath)
{
    // Read file into an array of lines with auto-detected line
endings
    $autodetect = ini_get('auto_detect_line_endings');
    ini_set('auto_detect_line_endings', '1');
    $lines = file($filePath, FILE_IGNORE_NEW_LINES | FILE_SKIP_E
MPTY_LINES);
    ini_set('auto_detect_line_endings', $autodetect);

    return $lines;
}

protected function isComment($line)
{
    return strpos(ltrim($line), '#') === 0;
}

protected function looksLikeSetter($line)
{
    return strpos($line, '=') !== false;
}
```

环境变量赋值是 `env` 文件加载的核心，主要由 `setEnvironmentVariable` 函数：

```
public function setEnvironmentVariable($name, $value = null)
{
    list($name, $value) = $this->normaliseEnvironmentVariable($n
ame, $value);

    if ($this->immutable && $this->getEnvironmentVariable($name)
 !== null) {
        return;
    }

    if (function_exists('apache_getenv') && function_exists('apa
che_setenv') && apache_getenv($name)) {
        apache_setenv($name, $value);
    }

    if (function_exists('putenv')) {
        putenv("$name=$value");
    }

    $_ENV[$name] = $value;
    $_SERVER[$name] = $value;
}
```

`normaliseEnvironmentVariable` 函数用来加载各种类型的环境变量：

```
protected function normaliseEnvironmentVariable($name, $value)
{
    list($name, $value) = $this->splitCompoundStringIntoParts($n
ame, $value);
    list($name, $value) = $this->sanitiseVariableName($name, $va
lue);
    list($name, $value) = $this->sanitiseVariableValue($name, $v
alue);

    $value = $this->resolveNestedVariables($value);

    return array($name, $value);
}
```

`splitCompoundStringIntoParts` 用于将赋值语句转化为环境变量名 `name` 和环境变量值 `value` 。

```
protected function splitCompoundStringIntoParts($name, $value)
{
    if (strpos($name, '=') !== false) {
        list($name, $value) = array_map('trim', explode('=', $na
me, 2));
    }

    return array($name, $value);
}
```

`sanitiseVariableName` 用于格式化环境变量名：

```
 protected function sanitiseVariableName($name, $value)
{
    $name = trim(str_replace(array('export ', '\'', '"'), '', $n
ame));

    return array($name, $value);
}
```

`sanitiseVariableValue` 用于格式化环境变量值：

```
protected function sanitiseVariableValue($name, $value)
{
    $value = trim($value);
    if (!$value) {
        return array($name, $value);
    }

    if ($this->beginsWithAQuote($value)) { // value starts with
a quote
        $quote = $value[0];
        $regexPattern = sprintf(
            '/^
            %1$s          # match a quote at the start of the va
```

```
lue
            (              # capturing sub-pattern used
             (?:           # we do not need to capture this
              [^%1$s\\\\] # any character other than a quote or
backslash
              |\\\\\\\\   # or two backslashes together
              |\\\\%1$s   # or an escaped quote e.g \"
             )*            # as many characters that match the pr
evious rules
            )              # end of the capturing sub-pattern
            %1$s           # and the closing quote
            .*$            # and discard any string after the clo
sing quote
            /mx',
            $quote
        );
        $value = preg_replace($regexPattern, '$1', $value);
        $value = str_replace("\\$quote", $quote, $value);
        $value = str_replace('\\\\', '\\', $value);
    } else {
        $parts = explode(' #', $value, 2);
        $value = trim($parts[0]);

        // Unquoted values cannot contain whitespace
        if (preg_match('/\s+/', $value) > 0) {
            throw new InvalidFileException('Dotenv values contai
ning spaces must be surrounded by quotes.');
        }
    }

    return array($name, trim($value));
}
```

这段代码是加载 `env` 文件最复杂的部分，我们详细来说：

- 若环境变量值是具体值，那么仅仅需要分割注释 `#` 部分，并判断是否存在空格符即可。
- 若环境变量值由引用构成，那么就需要进行正则匹配，具体的正则表达式为：

```
/^"((?:[^"\\]|\\\\|\\"))".*$/mx
```

这个正则表达式的意思是：

- 提取 `“”` 双引号内部的字符串，抛弃双引号之后的字符串
- 若双引号内部还有双引号，那么以最前面的双引号为提取内容，例如 "dfd("dfd")fdf"，我们只能提取出来最前面的部分 "dfd("
- 对于内嵌的引用可以使用 `\"` ，例如 "dfd\"dfd\"fdf",我们就可以提取出来 "dfd\"dfd\"fdf"。
- 不允许引用中含有 `\` ，但可以使用转义字符 `\\`

前言

本文主要介绍 `laravel` 加载 `config` 配置文件的相关源码。

config 配置文件的加载

config 配置文件由类 `\Illuminate\Foundation\Bootstrap\LoadConfiguration::class` 完成：

```
class LoadConfiguration
{
    public function bootstrap(Application $app)
    {
        $items = [];

        if (file_exists($cached = $app->getCachedConfigPath()))
{
            $items = require $cached;

            $loadedFromCache = true;
        }

        $app->instance('config', $config = new Repository($items
));

        if (! isset($loadedFromCache)) {
            $this->loadConfigurationFiles($app, $config);
        }

        $app->detectEnvironment(function () use ($config) {
            return $config->get('app.env', 'production');
        });

        date_default_timezone_set($config->get('app.timezone', '
UTC'));

        mb_internal_encoding('UTF-8');
    }
}
```

可以看到，配置文件的加载步骤：

- 加载缓存
- 若缓存不存在，则利用函数 `loadConfigurationFiles` 加载配置文件
- 加载环境变量、时间区、编码方式

函数 `loadConfigurationFiles` 用于加载配置文件：

```
protected function loadConfigurationFiles(Application $app, Repo
sitoryContract $repository)
{
    foreach ($this->getConfigurationFiles($app) as $key => $path
) {
        $repository->set($key, require $path);
    }
}
```

加载配置文件有两部分：搜索配置文件、加载配置文件的数组变量值

搜索配置文件

`getConfigurationFiles` 可以根据配置文件目录搜索所有的 `php` 为后缀的文件，并将其转化为 `files` 数组，其 `key` 为目录名以字符 `.` 为连接的字符串，`value` 为文件真实路径：

```
protected function getConfigurationFiles(Application $app)
{
    $files = [];

    $configPath = realpath($app->configPath());

    foreach (Finder::create()->files()->name('*.php')->in($confi
gPath) as $file) {
        $directory = $this->getNestedDirectory($file, $configPat
h);

        $files[$directory.basename($file->getRealPath(), '.php')
] = $file->getRealPath();
    }

    return $files;
}

protected function getNestedDirectory(SplFileInfo $file, $config
Path)
{
    $directory = $file->getPath();

    if ($nested = trim(str_replace($configPath, '', $directory),
 DIRECTORY_SEPARATOR)) {
        $nested = str_replace(DIRECTORY_SEPARATOR, '.', $nested).
'.';
    }

    return $nested;
}
```

加载配置文件数组

加载配置文件由类 `Illuminate\Config\Repository\LoadConfiguration` 完成：

```
class Repository
{
    public function set($key, $value = null)
    {
        $keys = is_array($key) ? $key : [$key => $value];

        foreach ($keys as $key => $value) {
            Arr::set($this->items, $key, $value);
        }
    }
}
```

加载配置文件时间上就是将所有配置文件的数值放入一个巨大的多维数组中，这一部分由类 `Illuminate\Support\Arr` 完成：

```
class Arr
{
    public static function set(&$array, $key, $value)
    {
        if (is_null($key)) {
            return $array = $value;
        }

        $keys = explode('.', $key);

        while (count($keys) > 1) {
            $key = array_shift($keys);

            if (! isset($array[$key]) || ! is_array($array[$key]
)) {
                $array[$key] = [];
            }

            $array = &$array[$key];
        }

        $array[array_shift($keys)] = $value;

        return $array;
    }
}
```

例如 `dir1.dir2.app` ,配置文件会生成 `$array[dir1][dir2][app]` 这样的数组。

配置文件数值的获取

当我们利用全局函数 `config` 来获取配置值的时候：

```
function config($key = null, $default = null)
{
    if (is_null($key)) {
        return app('config');
    }

    if (is_array($key)) {
        return app('config')->set($key);
    }

    return app('config')->get($key, $default);
}
```

配置文件的获取和加载类似，都是将字符串转为多维数组，然后获取具体数组值：

```
public static function get($array, $key, $default = null)
{
    if (! static::accessible($array)) {
        return value($default);
    }

    if (is_null($key)) {
        return $array;
    }

    if (static::exists($array, $key)) {
        return $array[$key];
    }

    foreach (explode('.', $key) as $segment) {
        if (static::accessible($array) && static::exists($array,
 $segment)) {
            $array = $array[$segment];
        } else {
            return value($default);
        }
    }

    return $array;
}
```

前言

对于一个优秀的框架来说，正确的异常处理可以防止暴露自身接口给用户，可以提供快速追溯问题的提示给开发人员。本文会详细的介绍 `laravel` 异常处理的源码。

PHP 异常处理

本章节参考 PHP错误异常处理详解。

异常处理（又称为错误处理）功能提供了处理程序运行时出现的错误或异常情况的方法。　　异常处理通常是防止未知错误产生所采取的处理措施。异常处理的好处是你不用再绞尽脑汁去考虑各种错误，这为处理某一类错误提供了一个很有效的方法，使编程效率大大提高。当异常被触发时，通常会发生：

- 当前代码状态被保存
- 代码执行被切换到预定义的异常处理器函数
- 根据情况，处理器也许会从保存的代码状态重新开始执行代码，终止脚本执行，或从代码中另外的位置继续执行脚本

PHP 5 提供了一种新的面向对象的错误处理方法。可以使用检测（try）、抛出（throw）和捕获（catch）异常。即使用try检测有没有抛出（throw）异常，若有异常抛出（throw），使用catch捕获异常。

一个 try 至少要有一个与之对应的 catch。定义多个 catch 可以捕获不同的对象。php 会按这些 catch 被定义的顺序执行，直到完成最后一个为止。而在这些 catch 内，又可以抛出新的异常。

异常的抛出

当一个异常被抛出时，其后的代码将不会继续执行，PHP 会尝试查找匹配的 `catch` 代码块。如果一个异常没有被捕获，而且又没用使用 `set_exception_handler()` 作相应的处理的话，那么 `PHP` 将会产生一个严重的错误，并且输出未能捕获异常 `(Uncaught Exception ... )` 的提示信息。

抛出异常，但不去捕获它：

```
ini_set('display_errors', 'On');
error_reporting(E_ALL & ~ E_WARNING);
$error = 'Always throw this error';
throw new Exception($error);
// 继续执行
echo 'Hello World';
```

上面的代码会获得类似这样的一个致命错误：

```
Fatal error: Uncaught exception 'Exception' with message 'Always
 throw this error' in E:\sngrep\index.php on line 5
Exception: Always throw this error in E:\sngrep\index.php on lin
e 5
Call Stack:
    0.0005     330680   1. {main}() E:\sngrep\index.php:0
```

Try, throw 和 catch

要避免上面这个致命错误，可以使用try catch捕获掉。

处理处理程序应当包括：

- Try - 使用异常的函数应该位于 "try" 代码块内。如果没有触发异常，则代码将照常继续执行。但是如果异常被触发，会抛出一个异常。
- Throw - 这里规定如何触发异常。每一个 "throw" 必须对应至少一个 "catch"
- Catch - "catch" 代码块会捕获异常，并创建一个包含异常信息的对象

抛出异常并捕获掉，可以继续执行后面的代码：

```
try {
    $error = 'Always throw this error';
    throw new Exception($error);

    // 从这里开始，tra 代码块内的代码将不会被执行
    echo 'Never executed';

} catch (Exception $e) {
    echo 'Caught exception: ',  $e->getMessage(),'<br>';
}

// 继续执行
echo 'Hello World';
```

顶层异常处理器 set_exception_handler

在我们实际开发中，异常捕捉仅仅靠 `try {} catch ()` 是远远不够的。`set_exception_handler()` 函数可设置处理所有未捕获异常的用户定义函数。

```
function myException($exception)
{
echo "<b>Exception:</b> " , $exception->getMessage();
}

set_exception_handler('myException');
throw new Exception('Uncaught Exception occurred');
```

扩展 PHP 内置的异常处理类

用户可以用自定义的异常处理类来扩展 PHP 内置的异常处理类。以下的代码说明了在内置的异常处理类中，哪些属性和方法在子类中是可访问和可继承的。

```
class Exception
{
    protected $message = 'Unknown exception';   // 异常信息
    protected $code = 0;                        // 用户自定义异常代
码
    protected $file;                            // 发生异常的文件名

    protected $line;                            // 发生异常的代码行
号

    function __construct($message = null, $code = 0);

    final function getMessage();                // 返回异常信息
    final function getCode();                   // 返回异常代码
    final function getFile();                   // 返回发生异常的文
件名
    final function getLine();                   // 返回发生异常的代
码行号
    final function getTrace();                  // backtrace()
数组
    final function getTraceAsString();          // 已格成化成字符串
的 getTrace() 信息

    /* 可重载的方法 */
    function __toString();                       // 可输出的字符串

}
```

如果使用自定义的类来扩展内置异常处理类，并且要重新定义构造函数的话，建议同时调用 `parent::__construct()` 来检查所有的变量是否已被赋值。当对象要输出字符串的时候，可以重载 `__toString()` 并自定义输出的样式。

```
class MyException extends Exception
{
    // 重定义构造器使 message 变为必须被指定的属性
    public function __construct($message, $code = 0) {
        // 自定义的代码

        // 确保所有变量都被正确赋值
        parent::__construct($message, $code);
    }

    // 自定义字符串输出的样式 */
    public function __toString() {
        return __CLASS__ . ": [{$this->code}]: {$this->message}\
n";
    }

    public function customFunction() {
        echo "A Custom function for this type of exception\n";
    }
}
```

`MyException` 类是作为旧的 exception 类的一个扩展来创建的。这样它就继承了旧类的所有属性和方法，我们可以使用 `exception` 类的方法，比如 `getLine()` 、 `getFile()` 以及 `getMessage()` 。

PHP 错误处理

PHP 的错误级别

值	常量	说明
1	E_ERROR	致命的运行时错误。这类错误一般是不可恢复的情况，例如内存分配导致的问题。后果是导致脚本终止不再继续运行。
2	E_WARNING	运行时警告 (非致命错误)。仅给出提示信息，但是脚本不会终止运行。

4	E_PARSE	编译时语法解析错误。解析错误仅仅由分析器产生。
8	E_NOTICE	运行时通知。表示脚本遇到可能会表现为错误的情况，但是在可以正常运行的脚本里面也可能会有类似的通知。
16	E_CORE_ERROR	在PHP初始化启动过程中发生的致命错误。该错误类似 E_ERROR，但是是由PHP引擎核心产生的。
32	E_CORE_WARNING	PHP初始化启动过程中发生的警告(非致命错误) 。类似 E_WARNING，但是是由PHP引擎核心产生的。
64	E_COMPILE_ERROR	致命编译时错误。类似E_ERROR, 但是是由Zend脚本引擎产生的。
128	E_COMPILE_WARNING	编译时警告 (非致命错误)。类似 E_WARNING，但是是由Zend脚本引擎产生的。
256	E_USER_ERROR	用户产生的错误信息。类似 E_ERROR, 但是是由用户自己在代码中使用PHP函数 trigger_error()来产生的。
512	E_USER_WARNING	用户产生的警告信息。类似 E_WARNING, 但是是由用户自己在代码中使用PHP函数 trigger_error()来产生的。
1024	E_USER_NOTICE	用户产生的通知信息。类似 E_NOTICE, 但是是由用户自己在代码中使用PHP函数 trigger_error()来产生的。
2048	E_STRICT	启用 PHP 对代码的修改建议，以确保代码具有最佳的互操作性和向前兼容性。
4096	E_RECOVERABLE_ERROR	可被捕捉的致命错误。 它表示发生了一个可能非常危险的错误，但是还没有导致PHP引擎处于不稳定的状态。 如果该错误没有被用户自定义句柄捕获 (参见 set_error_handler())，将成为一个 E_ERROR　从而脚本会终止运行。
		运行时通知。启用后将会对在未来版

8192	E_DEPRECATED	本中可能无法正常工作的代码给出警告。
16384	E_USER_DEPRECATED	用户产少的警告信息。 类似 E_DEPRECATED, 但是是由用户自己在代码中使用PHP函数 trigger_error()来产生的。
30719	E_ALL	用户产少的警告信息。 类似 E_DEPRECATED, 但是是由用户自己在代码中使用PHP函数 trigger_error()来产生的。

错误的抛出

除了系统在运行 php 代码抛出的意外错误。我们还可以利用 `rigger_error` 产生一个自定义的用户级别的 `error/warning/notice` 错误信息:

```
if ($divisor == 0) {
    trigger_error("Cannot divide by zero", E_USER_ERROR);
}
```

顶级错误处理器

顶级错误处理器 `set_error_handler` 一般用于捕捉 `E_NOTICE` 、`E_USER_ERROR` 、`E_USER_WARNING` 、`E_USER_NOTICE` 级别的错误，不能捕捉 `E_ERROR` , `E_PARSE` , `E_CORE_ERROR` , `E_CORE_WARNING` , `E_COMPILE_ERROR` 和 `E_COMPILE_WARNING` 。

`register_shutdown_function`

`register_shutdown_function()` 函数可实现当程序执行完成后执行的函数，其功能为可实现程序执行完成的后续操作。程序在运行的时候可能存在执行超时，或强制关闭等情况，但这种情况下默认的提示是非常不友好的，如果使用 `register_shutdown_function()` 函数捕获异常，就能提供更加友好的错误展示方式，同时可以实现一些功能的后续操作，如执行完成后的临时数据清理，包括临时文件等。

可以这样理解调用条件：

- 当页面被用户强制停止时
- 当程序代码运行超时时
- 当ＰＨＰ代码执行完成时，代码执行存在异常和错误、警告

我们前面说过，`set_error_handler` 能够捕捉的错误类型有限，很多致命错误例如解析错误等都无法捕捉，但是这类致命错误发生时，PHP 会调用 `register_shutdown_function` 所注册的函数，如果结合函数 `error_get_last` ，就会获取错误发生的信息。

Laravel 异常处理

`laravel` 的异常处理由类 `\Illuminate\Foundation\Bootstrap\HandleExceptions::class` 完成：

```
class HandleExceptions
{
    public function bootstrap(Application $app)
    {
        $this->app = $app;

        error_reporting(-1);

        set_error_handler([$this, 'handleError']);

        set_exception_handler([$this, 'handleException']);

        register_shutdown_function([$this, 'handleShutdown']);

        if (! $app->environment('testing')) {
            ini_set('display_errors', 'Off');
        }
    }
}
```

异常转化

`laravel` 的异常处理均由函数 `handleException` 负责。

`PHP7` 实现了一个全局的 `throwable` 接口，原来的 `Exception` 和部分 `Error` 都实现了这个接口， 以接口的方式定义了异常的继承结构。于是， `PHP7` 中更多的 `Error` 变为可捕获的 `Exception` 返回给开发者，如果不进行捕获则为 `Error` ，如果捕获就变为一个可在程序内处理的 `Exception` 。这些可被捕获的 `Error` 通常都是不会对程序造成致命伤害的 `Error` ，例如函数不存在。

`PHP7` 中，基于 `/Error exception` ，派生了5个新的engine exception： `ArithmeticError` / `AssertionError` / `DivisionByZeroError` / `ParseError` / `TypeError` 。在 `PHP7` 里，无论是老的 `/Exception` 还是新的 `/Error` ，它们都实现了一个共同的interface: `/Throwable` 。

因此，遇到非 `Exception` 类型的异常，首先就要将其转化为 `FatalThrowableError` 类型：

```
public function handleException($e)
{
    if (! $e instanceof Exception) {
        $e = new FatalThrowableError($e);
    }

    $this->getExceptionHandler()->report($e);

    if ($this->app->runningInConsole()) {
        $this->renderForConsole($e);
    } else {
        $this->renderHttpResponse($e);
    }
}
```

`FatalThrowableError` 是 `Symfony` 继承 `\ErrorException` 的错误异常类：

```
class FatalThrowableError extends FatalErrorException
{
    public function __construct(\Throwable $e)
    {
        if ($e instanceof \ParseError) {
            $message = 'Parse error: '.$e->getMessage();
            $severity = E_PARSE;
        } elseif ($e instanceof \TypeError) {
            $message = 'Type error: '.$e->getMessage();
            $severity = E_RECOVERABLE_ERROR;
        } else {
            $message = $e->getMessage();
            $severity = E_ERROR;
        }

        \ErrorException::__construct(
            $message,
            $e->getCode(),
            $severity,
            $e->getFile(),
            $e->getLine()
        );

        $this->setTrace($e->getTrace());
    }
}
```

异常 Log

当遇到异常情况的时候，`laravel` 首要做的事情就是记录 `log`，这个就是 `report` 函数的作用。

```
 protected function getExceptionHandler()
{
    return $this->app->make(ExceptionHandler::class);
}
```

`laravel` 在 `Ioc` 容器中默认的异常处理类是 `Illuminate\Foundation\Exceptions\Handler`：

```
class Handler implements ExceptionHandlerContract
{
    public function report(Exception $e)
    {
        if ($this->shouldntReport($e)) {
            return;
        }

        try {
            $logger = $this->container->make(LoggerInterface::cl
ass);
        } catch (Exception $ex) {
            throw $e; // throw the original exception
        }

        $logger->error($e);
    }

    protected function shouldntReport(Exception $e)
    {
        $dontReport = array_merge($this->dontReport, [HttpRespon
seException::class]);

        return ! is_null(collect($dontReport)->first(function ($
type) use ($e) {
            return $e instanceof $type;
        }));
    }
}
```

异常页面展示

记录 `log` 后，就要将异常转化为页面向开发者展示异常的信息，以便查看问题的来源：

```
protected function renderHttpResponse(Exception $e)
{
    $this->getExceptionHandler()->render($this->app['request'],
$e)->send();
}

class Handler implements ExceptionHandlerContract
{
    public function render($request, Exception $e)
    {
        $e = $this->prepareException($e);

        if ($e instanceof HttpResponseException) {
            return $e->getResponse();
        } elseif ($e instanceof AuthenticationException) {
            return $this->unauthenticated($request, $e);
        } elseif ($e instanceof ValidationException) {
            return $this->convertValidationExceptionToResponse($
e, $request);
        }

        return $this->prepareResponse($request, $e);
    }
}
```

对于不同的异常， `laravel` 有不同的处理，大致有 `HttpException` 、 `HttpResponseException` 、 `AuthorizationException` 、 `ModelNotFoundException` 、 `AuthenticationException` 、 `ValidationException` 。由于特定的不同异常带有自身的不同需求，本文不会特别介绍。本文继续介绍最普通的异常 `HttpException` 的处理：

```
protected function prepareResponse($request, Exception $e)
{
    if ($this->isHttpException($e)) {
        return $this->toIlluminateResponse($this->renderHttpExce
ption($e), $e);
    } else {
        return $this->toIlluminateResponse($this->convertExcepti
onToResponse($e), $e);
    }
}

protected function renderHttpException(HttpException $e)
{
    $status = $e->getStatusCode();

    view()->replaceNamespace('errors', [
        resource_path('views/errors'),
        __DIR__.'/views',
    ]);

    if (view()->exists("errors::{$status}")) {
        return response()->view("errors::{$status}", ['exception'
 => $e], $status, $e->getHeaders());
    } else {
        return $this->convertExceptionToResponse($e);
    }
}
```

对于 `HttpException` 来说，会根据其错误的状态码，选取不同的错误页面模板，若不存在相关的模板，则会通过 `SymfonyResponse` 来构造异常展示页面：

```
protected function convertExceptionToResponse(Exception $e)
{
    $e = FlattenException::create($e);

    $handler = new SymfonyExceptionHandler(config('app.debug'));

    return SymfonyResponse::create($handler->getHtml($e), $e->ge
tStatusCode(), $e->getHeaders());
}

protected function toIlluminateResponse($response, Exception $e)
{
    if ($response instanceof SymfonyRedirectResponse) {
        $response = new RedirectResponse($response->getTargetUrl
(), $response->getStatusCode(), $response->headers->all());
    } else {
        $response = new Response($response->getContent(), $respo
nse->getStatusCode(), $response->headers->all());
    }

    return $response->withException($e);
}
```

laravel 错误处理

```
public function handleError($level, $message, $file = '', $line
= 0, $context = [])
{
    if (error_reporting() & $level) {
        throw new ErrorException($message, 0, $level, $file, $li
ne);
    }
}

public function handleShutdown()
{
    if (! is_null($error = error_get_last()) && $this->isFatal($
error['type'])) {
        $this->handleException($this->fatalExceptionFromError($e
rror, 0));
    }
}

protected function fatalExceptionFromError(array $error, $traceO
ffset = null)
{
    return new FatalErrorException(
        $error['message'], $error['type'], 0, $error['file'], $e
rror['line'], $traceOffset
    );
}

protected function isFatal($type)
{
    return in_array($type, [E_COMPILE_ERROR, E_CORE_ERROR, E_ERR
OR, E_PARSE]);
}
```

对于不致命的错误，例如 `notice` 级别的错误， `handleError` 即可截取， `laravel` 将错误转化为了异常，交给了 `handleException` 去处理。

对于致命错误，例如 `E_PARSE` 解析错误， `handleShutdown` 将会启动，并且判断当前脚本结束是否是由于致命错误，如果是致命错误，将会将其转化为 `FatalErrorException` , 交给了 `handleException` 作为异常去处理。

前言

服务提供者是 `laravel` 框架的重要组成部分，承载着各种服务，自定义的应用以及所有 `Laravel` 的核心服务都是通过服务提供者启动。本文将会介绍服务提供者的源码分析，关于服务提供者的使用，请参考官方文档 ：服务提供者。

服务提供者的注册

服务提供者的启动由类 `\Illuminate\Foundation\Bootstrap\RegisterProviders::class` 负责，该类用于加载所有服务提供者的 `register` 函数，并保存延迟加载的服务的信息，以便实现延迟加载。

```
class RegisterProviders
{
    public function bootstrap(Application $app)
    {
        $app->registerConfiguredProviders();
    }
}

class Application extends Container implements ApplicationContra
ct, HttpKernelInterface
{
    public function registerConfiguredProviders()
    {
        (new ProviderRepository($this, new Filesystem, $this->ge
tCachedServicesPath()))
                    ->load($this->config['app.providers']);
    }

    public function getCachedServicesPath()
    {
        return $this->bootstrapPath().'/cache/services.php';
    }
}
```

以上可以看出，所有服务提供者都在配置文件 `app.php` 文件的 `providers` 数组中。类 `ProviderRepository` 负责所有的服务加载功能：

```
class ProviderRepository
{
    public function load(array $providers)
    {
        $manifest = $this->loadManifest();

        if ($this->shouldRecompile($manifest, $providers)) {
            $manifest = $this->compileManifest($providers);
        }

        foreach ($manifest['when'] as $provider => $events) {
            $this->registerLoadEvents($provider, $events);
        }

        foreach ($manifest['eager'] as $provider) {
            $this->app->register($provider);
        }

        $this->app->addDeferredServices($manifest['deferred']);
    }
}
```

加载服务缓存文件

`laravel` 会把所有的服务整理起来，作为缓存写在缓存文件中：

```
return array (
    'providers' =>
    array (
      0 => 'Illuminate\\Auth\\AuthServiceProvider',
      1 => 'Illuminate\\Broadcasting\\BroadcastServiceProvider',
      ...
    ),

    'eager' =>
    array (
      0 => 'Illuminate\\Auth\\AuthServiceProvider',
      1 => 'Illuminate\\Cookie\\CookieServiceProvider',
      ...
    ),

    'deferred' =>
    array (
      'Illuminate\\Broadcasting\\BroadcastManager' => 'Illuminat
e\\Broadcasting\\BroadcastServiceProvider',
      'Illuminate\\Contracts\\Broadcasting\\Factory' => 'Illumin
ate\\Broadcasting\\BroadcastServiceProvider',
      ...
    ),

    'when' =>
    array (
      'Illuminate\\Broadcasting\\BroadcastServiceProvider' =>
      array (
      ),
      ...
    ),
```

- 缓存文件中 `providers` 放入了所有自定义和框架核心的服务。
- `eager` 数组中放入了所有需要立即启动的服务提供者。
- `deferred` 数组中放入了所有需要延迟加载的服务提供者。
- `when` 放入了延迟加载需要激活的事件。

加载服务提供者缓存文件：

```
public function loadManifest()
{
    if ($this->files->exists($this->manifestPath)) {
        $manifest = $this->files->getRequire($this->manifestPath
);

        if ($manifest) {
            return array_merge(['when' => []], $manifest);
        }
    }
}
```

编译服务提供者

若 `laravel` 中的服务提供者没有缓存文件或者有变动，那么就会重新生成缓存文件：

```
public function shouldRecompile($manifest, $providers)
{
    return is_null($manifest) || $manifest['providers'] != $prov
iders;
}

protected function compileManifest($providers)
{
    $manifest = $this->freshManifest($providers);

    foreach ($providers as $provider) {
        $instance = $this->createProvider($provider);

        if ($instance->isDeferred()) {
            foreach ($instance->provides() as $service) {
                $manifest['deferred'][$service] = $provider;
            }

            $manifest['when'][$provider] = $instance->when();
        }

        else {
            $manifest['eager'][] = $provider;
        }
    }

    return $this->writeManifest($manifest);
}

protected function freshManifest(array $providers)
{
    return ['providers' => $providers, 'eager' => [], 'deferred'
 => []];
}
```

- 如果服务提供者是需要立即注册的，那么将会放入缓存文件中 `eager` 数组中。
- 如果服务提供者是延迟加载的，那么其函数 `provides()` 通常会提供服务别名，这个服务别名通常是向服务容器中注册的别名，别名将会放入缓存文件的

  `deferred` 数组中。
- 延迟加载若有 `event` 事件激活，那么可以在 `when` 函数中写入事件类，并写入缓存文件的 `when` 数组中。

延迟服务提供者事件注册

延迟服务提供者除了利用 `IOC` 容器解析服务方式激活，还可以利用 `Event` 事件来激活：

```
protected function registerLoadEvents($provider, array $events)
{
    if (count($events) < 1) {
        return;
    }

    $this->app->make('events')->listen($events, function () use
($provider) {
        $this->app->register($provider);
    });
}
```

注册即时启动的服务提供者

服务提供者的注册函数 `register()` 由类 `Application` 来调用：

```
class Application extends Container implements ApplicationContra
ct, HttpKernelInterface
{
    public function register($provider, $options = [], $force =
false)
    {
        if (($registered = $this->getProvider($provider)) && ! $
force) {
            return $registered;
        }

        if (is_string($provider)) {
```

```
            $provider = $this->resolveProvider($provider);
        }

        if (method_exists($provider, 'register')) {
            $provider->register();
        }

        $this->markAsRegistered($provider);

        if ($this->booted) {
            $this->bootProvider($provider);
        }

        return $provider;
    }

    public function getProvider($provider)
    {
        $name = is_string($provider) ? $provider : get_class($pr
ovider);

        return Arr::first($this->serviceProviders, function ($va
lue) use ($name) {
            return $value instanceof $name;
        });
    }

    public function resolveProvider($provider)
    {
        return new $provider($this);
    }

    protected function markAsRegistered($provider)
    {
        $this->serviceProviders[] = $provider;

        $this->loadedProviders[get_class($provider)] = true;
    }

    protected function bootProvider(ServiceProvider $provider)
```

```
    {
        if (method_exists($provider, 'boot')) {
            return $this->call([$provider, 'boot']);
        }
    }
}
```

可以看出，服务提供者的注册过程：

- 判断当前服务提供者是否被注册过，如注册过直接返回对象
- 解析服务提供者
- 调用服务提供者的 `register` 函数
- 标记当前服务提供者已经注册完毕
- 若框架已经加载注册完毕所有的服务容器，那么就启动服务提供者的 `boot` 函数，该函数由于是 `call` 调用，所以支持依赖注入。

延迟服务提供者激活与注册

延迟服务提供者首先需要添加到 `Application` 中：

```
public function addDeferredServices(array $services)
{
    $this->deferredServices = array_merge($this->deferredService
s, $services);
}
```

我们之前说过，延迟服务提供者的激活注册有两种方法：事件与服务解析。

当特定的事件被激发后，就会调用 `Application` 的 `register` 函数，进而调用服务提供者的 `register` 函数，实现服务的注册。

当利用 `Ioc` 容器解析服务名时，例如解析服务名 `BroadcastingFactory`：

```
class BroadcastServiceProvider extends ServiceProvider
{
    protected $defer = true;

    public function provides()
    {
        return [
            BroadcastManager::class,
            BroadcastingFactory::class,
            BroadcasterContract::class,
        ];
    }
}
```

```
public function make($abstract)
{
    $abstract = $this->getAlias($abstract);

    if (isset($this->deferredServices[$abstract])) {
        $this->loadDeferredProvider($abstract);
    }

    return parent::make($abstract);
}

public function loadDeferredProvider($service)
{
    if (! isset($this->deferredServices[$service])) {
        return;
    }

    $provider = $this->deferredServices[$service];

    if (! isset($this->loadedProviders[$provider])) {
        $this->registerDeferredProvider($provider, $service);
    }
}
```

由 `deferredServices` 数组可以得知，`BroadcastingFactory` 为延迟服务，接着程序会利用函数 `loadDeferredProvider` 来加载延迟服务提供者，调用服务提供者的 `register` 函数，若当前的框架还未注册完全部服务。那么将会放入服务启动的回调函数中，以待服务启动时调用：

```
public function registerDeferredProvider($provider, $service = n
ull)
{
    if ($service) {
        unset($this->deferredServices[$service]);
    }

    $this->register($instance = new $provider($this));

    if (! $this->booted) {
        $this->booting(function () use ($instance) {
            $this->bootProvider($instance);
        });
    }
}
```

关于服务提供者的注册函数：

```
class BroadcastServiceProvider extends ServiceProvider
{
    protected $defer = true;

    public function register()
    {
        $this->app->singleton(BroadcastManager::class, function
($app) {
            return new BroadcastManager($app);
        });

        $this->app->singleton(BroadcasterContract::class, functi
on ($app) {
            return $app->make(BroadcastManager::class)->connecti
on();
        });

        $this->app->alias(
            BroadcastManager::class, BroadcastingFactory::class
        );
    }

    public function provides()
    {
        return [
            BroadcastManager::class,
            BroadcastingFactory::class,
            BroadcasterContract::class,
        ];
    }
}
```

函数 `register` 为类 `BroadcastingFactory` 向 `Ioc` 容器绑定了特定的实现类 `BroadcastManager` ，这样 `Ioc` 容器中的 `make` 函数：

```
public function make($abstract)
{
    $abstract = $this->getAlias($abstract);

    if (isset($this->deferredServices[$abstract])) {
        $this->loadDeferredProvider($abstract);
    }

    return parent::make($abstract);
}
```

`parent::make($abstract)` 就会正确的解析服务 `BroadcastingFactory` 。

因此函数 `provides()` 返回的元素一定都是 `register()` 向 `IOC` 容器中绑定的类名或者别名。这样当我们利用服务容器来利用 `App::make()` 解析这些类名的时候，服务容器才会根据服务提供者的 `register()` 函数中绑定的实现类，从而正确解析服务功能。

服务容器的启动

服务容器的启动由类 `\Illuminate\Foundation\Bootstrap\BootProviders::class` 负责：

```
class BootProviders
{
    public function bootstrap(Application $app)
    {
        $app->boot();
    }
}

class Application extends Container implements ApplicationContra
ct, HttpKernelInterface
{
    public function boot()
    {
        if ($this->booted) {
            return;
        }

        $this->fireAppCallbacks($this->bootingCallbacks);

        array_walk($this->serviceProviders, function ($p) {
            $this->bootProvider($p);
        });

        $this->booted = true;

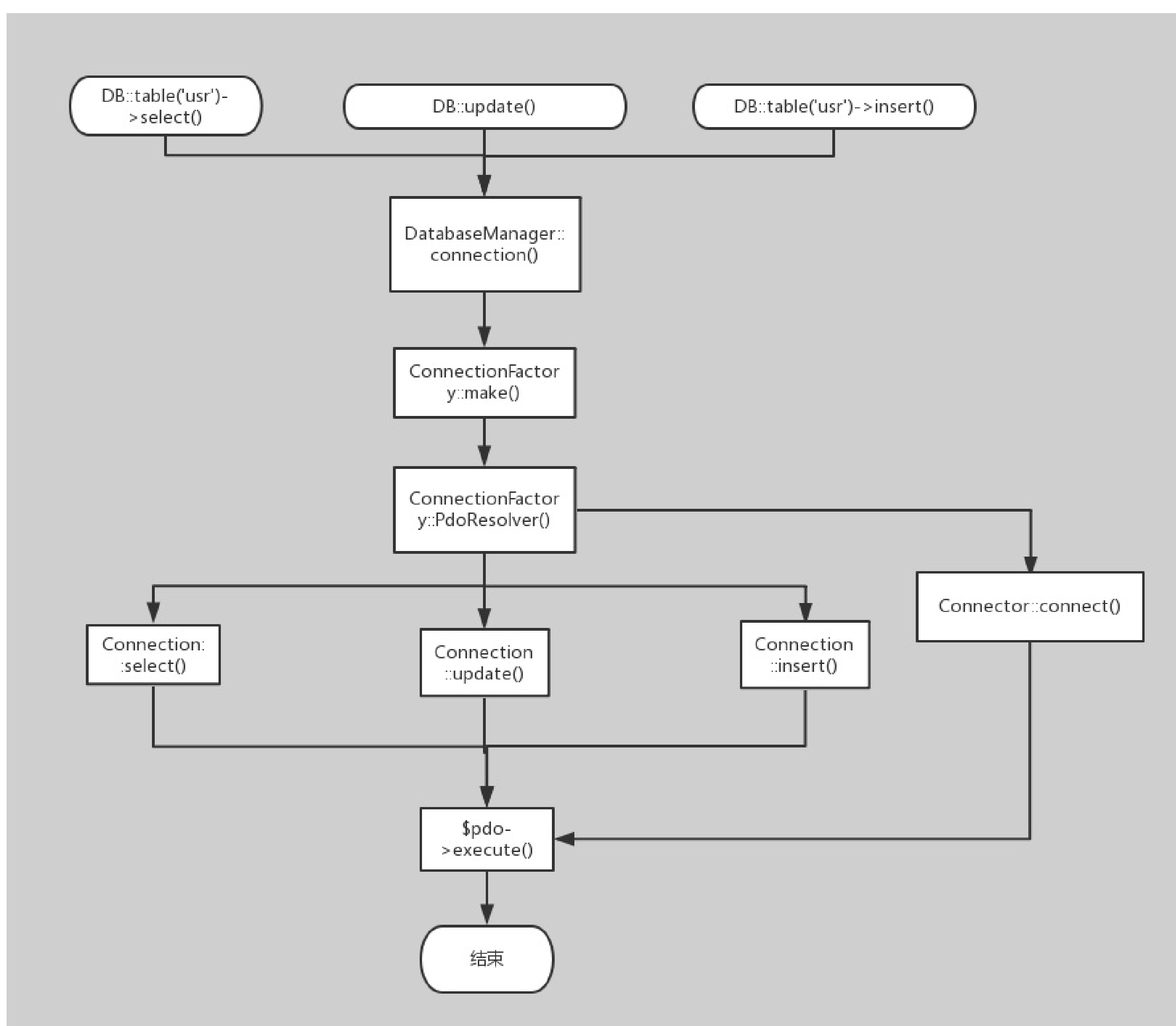
        $this->fireAppCallbacks($this->bootedCallbacks);
    }

    protected function bootProvider(ServiceProvider $provider)
    {
        if (method_exists($provider, 'boot')) {
            return $this->call([$provider, 'boot']);
        }
    }
}
```

前言

数据库是 `laravel` 及其重要的组成部分，大致的讲，`laravel` 的数据库功能可以分为两部分：数据库 `DB`、数据库 `Eloquent Model`。数据库的 `Eloquent` 是功能十分丰富的 `ORM`，让我们可以避免写繁杂的 `sql` 语句。数据库 `DB` 是比较底层的与 `pdo` 交互的功能，`Eloquent` 的底层依赖于 `DB`。本文将会介绍数据库 `DB` 中关于数据库服务的启动与连接部分。

在详细讲解数据库各个功能之前，我们先看看支撑着整个 `laravel` 数据库功能的框架：



- `DB` 也就是 `DatabaseManager`，承担着数据库接口的工作，一切数据库相关的操作，例如查询、更新、插入、删除都可以通过 `DB` 这个接口来完成。但是，具体的调用 `pdo` API 的工作却不是由该类完成的，它仅仅是一个对外的接口而已。

- `ConnectionFactory` 顾名思义专门为 `DB` 构造初始化 `connector` 、 `connection` 对象，
- `connector` 负责数据库的连接功能，为保障程序的高效， `laravel` 将其包装成为闭包函数，并将闭包函数作为 `connection` 的一个成员对象，实现懒加载。
- `connection` 负责数据库的具体功能，负责底层与 `pdo` API 的交互。

数据库服务的注册与启动

数据库服务也是一种服务提供者： `Illuminate\Database\DatabaseServiceProvider`

```
public function register()
{
    Model::clearBootedModels();

    $this->registerConnectionServices();

    $this->registerEloquentFactory();

    $this->registerQueueableEntityResolver();
}
```

我们先来看这个注册函数的第一句: `Model::clearBootedModels()` 。这一句其实是为了 `Eloquent` 服务的启动做准备。数据库的 `Eloquent Model` 有一个静态的成员变量数组 `$booted` ，这个静态数组存储了所有已经被初始化的数据库 `model` ，以便加载数据库模型时更加迅速。因此，在 `Eloquent` 服务启动之前需要初始化静态成员变量 `$booted` ：

```
public static function clearBootedModels()
{
    static::$booted = [];

    static::$globalScopes = [];
}
```

接下来我们就开始看数据库服务的注册最重要的两部分： `ConnectionServices` 与 `Eloquent` .

ConnectionServices 注册

```
protected function registerConnectionServices()
{
    $this->app->singleton('db.factory', function ($app) {
        return new ConnectionFactory($app);
    });

   $this->app->singleton('db', function ($app) {
        return new DatabaseManager($app, $app['db.factory']);
    });

    $this->app->bind('db.connection', function ($app) {
        return $app['db']->connection();
    });
}
```

可以看出，数据库服务向 `IOC` 容器注册了 `db` 、 `db.factory` 与 `db.connection` 。

- 最重要的莫过于 `db` 对象，它有一个 `Facade` 是 `DB` ， 我们可以利用 `DB::connection()` 来连接任意数据库，可以利用 `DB::select()` 来进行数据库的查询，可以说 `DB` 就是我们操作数据库的接口。
- `db.factory` 负责为 `DB` 创建 `connector` 提供数据库的底层连接服务，负责为 `DB` 创建 `connection` 对象来进行数据库的查询等操作。
- `db.connection` 是 `laravel` 用于与数据库 `pdo` 接口进行交互的底层类，可用于数据库的查询、更新、创建等操作。

Eloquent 注册

```
protected function registerEloquentFactory()
{
    $this->app->singleton(FakerGenerator::class, function () {
        return FakerFactory::create();
    });

    $this->app->singleton(EloquentFactory::class, function ($app)
 {
        return EloquentFactory::construct(
            $app->make(FakerGenerator::class), database_path('fa
ctories')
        );
    });
}
```

`EloquentFactory` 用于创建 `Eloquent Model`，用于全局函数 `factory()` 来创建数据库模型。

数据库服务的启动

```
public function boot()
{
    Model::setConnectionResolver($this->app['db']);

    Model::setEventDispatcher($this->app['events']);
}
```

数据库服务的启动主要设置 `Eloquent Model` 的 `connection resolver`，用于数据库模型 `model` 利用 `db` 来来连接数据库。还有设置数据库事件的分发器 `dispatcher`，用于监听数据库的事件。

DatabaseManager——数据库的接口

如果我们想要使用任何数据库服务，首先要做的事情当然是利用用户名与密码来连接数据库。在 `laravel` 中，数据库的用户名与密码一般放在 `.env` 文件中或者放入 `nginx` 配置中，并且利用数据库的接口 `DB` 来与 `pdo` 进行交互，利用 `pdo` 来连接数据库。

`DB` 即是类 `Illuminate\Database\DatabaseManager`，首先我们来看看其构造函数：

```
public function __construct($app, ConnectionFactory $factory)
{
    $this->app = $app;
    $this->factory = $factory;
}
```

我们称 `DB` 为一个接口，或者是一个门面模式，是因为数据库操作，例如数据库的连接或者查询、更新等操作均不是 `DB` 的功能，数据库的连接使用类 `Illuminate\Database\Connectors\Connector` 完成，数据库的查询等操作由类 `Illuminate\Database\Connection` 完成，

因此，我们不必直接操作 `connector` 或者 `connection`，仅仅会操作 `DB` 即可。

那么 `DB` 是如何实现 `connector` 或者 `connection` 的功能的呢？关键还是这个 `ConnectionFactory` 类，这个工厂类专门为 `DB` 来生成 `connection` 对象，并将其放入 `DB` 的成员变量数组 `$connections` 中去。`connection` 中会包含 `connector` 对象来实现数据库的连接工作。

```
class DatabaseManager implements ConnectionResolverInterface
{
    protected $app;
    protected $factory;
    protected $connections = [];

    public function __call($method, $parameters)
    {
        return $this->connection()->$method(...$parameters);
    }
}
```

魔术函数实现了 `DB` 与 `connection` 的无缝连接，任何对数据库的操作，例如 `DB::select()` 、 `DB::table('user')->save()` ，都会被转移至 `connection` 中去。

connection 函数——获取数据库连接对象

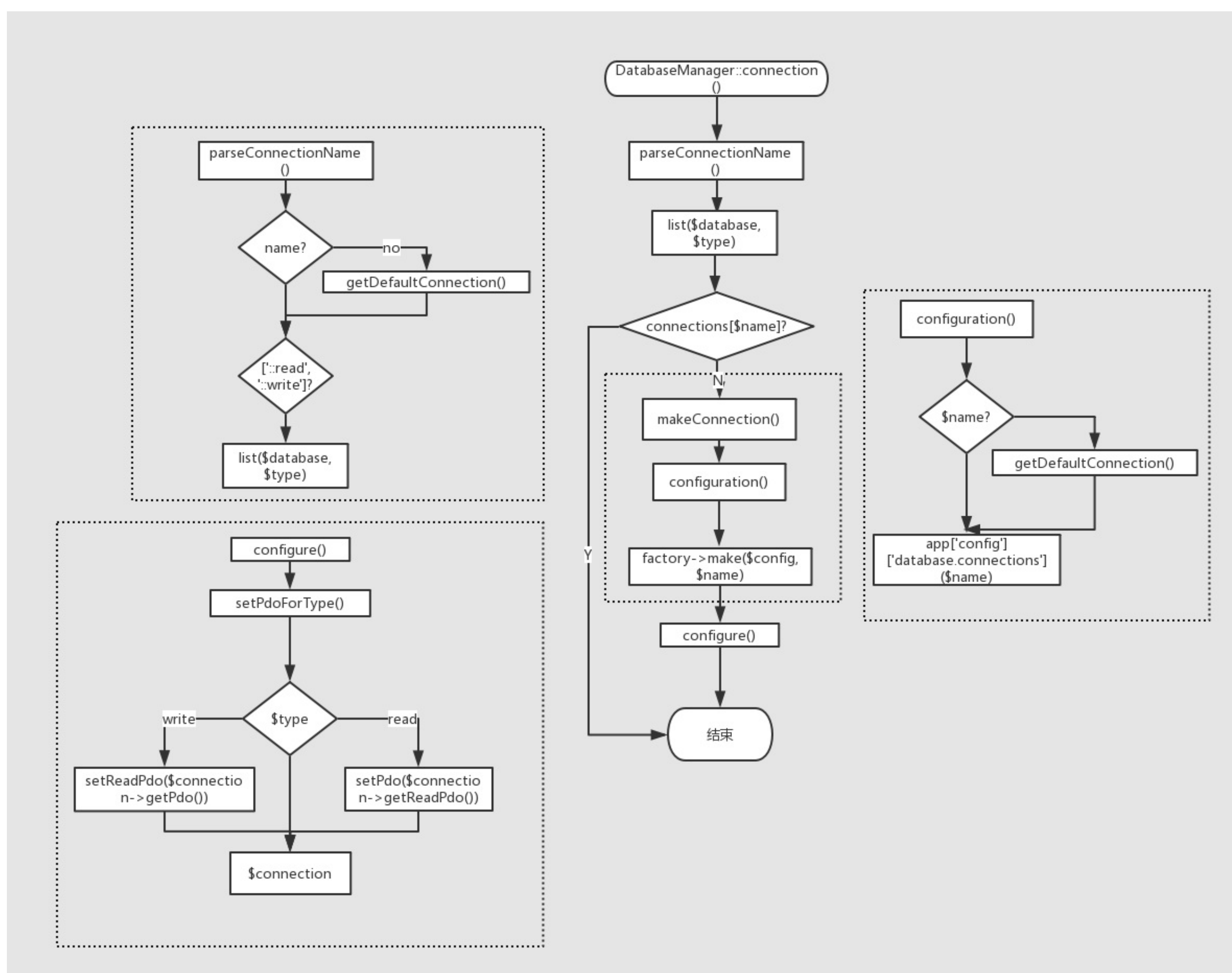
```
public function connection($name = null)
{
    list($database, $type) = $this->parseConnectionName($name);

    $name = $name ?: $database;

    if (! isset($this->connections[$name])) {
        $this->connections[$name] = $this->configure(
                $connection = $this->makeConnection($database),
$type
        );
    }

    return $this->connections[$name];
}
```

具体流程如下：



`DB` 的 `connection` 函数可以传入数据库的名字，也可以不传任何参数，此时会连接默认数据库，默认数据库的设置在 `config/database` 文件中。

`connection` 函数流程：

- 解析数据库名称与数据库类型，例如只读、写
- 若没有创建过与该数据库的连接，则开始创建数据库连接
- 返回数据库连接对象 `connection`

```
protected function parseConnectionName($name)
{
    $name = $name ?: $this->getDefaultConnection();

    return Str::endsWith($name, ['::read', '::write'])
                        ? explode('::', $name, 2) : [$name, null
];
}

public function getDefaultConnection()
{
    return $this->app['config']['database.default'];
}
```

可以看出，若没有特别指定连接的数据库名称，那么就会利用文件 `config/database` 文件中设置的 `default` 数据库名称作为默认连接数据库名称。若数据库支持读写分离，那么还可以指定数据库的读写属性，例如 `mysql::read` 。

makeConnection 函数——创建新的数据库连接对象

当框架从未连接过当前数据库的时候，就要对数据库进行连接操作，首先程序会调用 `makeConnection` 函数：

```
protected function makeConnection($name)
{
    $config = $this->configuration($name);

    if (isset($this->extensions[$name])) {
        return call_user_func($this->extensions[$name], $config,
 $name);
    }

    if (isset($this->extensions[$driver = $config['driver']])) {
        return call_user_func($this->extensions[$driver], $confi
g, $name);
    }

    return $this->factory->make($config, $name);
}
```

可以看出，连接数据库仅仅需要两个步骤：获取数据库配置、利用 `connection factory` 获取 `connection` 对象。

获取数据库配置：

```
protected function configuration($name)
{
    $name = $name ?: $this->getDefaultConnection();

    $connections = $this->app['config']['database.connections'];

    if (is_null($config = Arr::get($connections, $name))) {
        throw new InvalidArgumentException("Database [$name] not
 configured.");
    }

    return $config;
}
```

也是非常简单，直接从配置文件中获取当前数据库的配置：

```
'connections' => [
    'mysql' => [
        'driver' => 'mysql',
        'host' => env('DB_HOST', '127.0.0.1'),
        'port' => env('DB_PORT', '3306'),
        'database' => env('DB_DATABASE', 'forge'),
        'username' => env('DB_USERNAME', 'forge'),
        'password' => env('DB_PASSWORD', ''),
        'charset' => 'utf8mb4',
        'collation' => 'utf8mb4_unicode_ci',
        'prefix' => '',
        'strict' => true,
        'engine' => null,
        'read' => [
            'database'  => env('DB_DATABASE', 'forge'),
        ],
        'write' => [
            'database'  => env('DB_DATABASE', 'forge'),
        ],
    ],
],
```

`$this->factory->make($config, $name)` 函数向我们提供了数据库连接对象。

configure——连接对象读写配置

当我们从 `connection factory` 中获取到连接对象 `connection` 之后，我们就要根据传入的参数进行读写配置：

```
protected function configure(Connection $connection, $type)
{
    $connection = $this->setPdoForType($connection, $type);

    if ($this->app->bound('events')) {
        $connection->setEventDispatcher($this->app['events']);
    }

    $connection->setReconnector(function ($connection) {
        $this->reconnect($connection->getName());
    });

    return $connection;
}
```

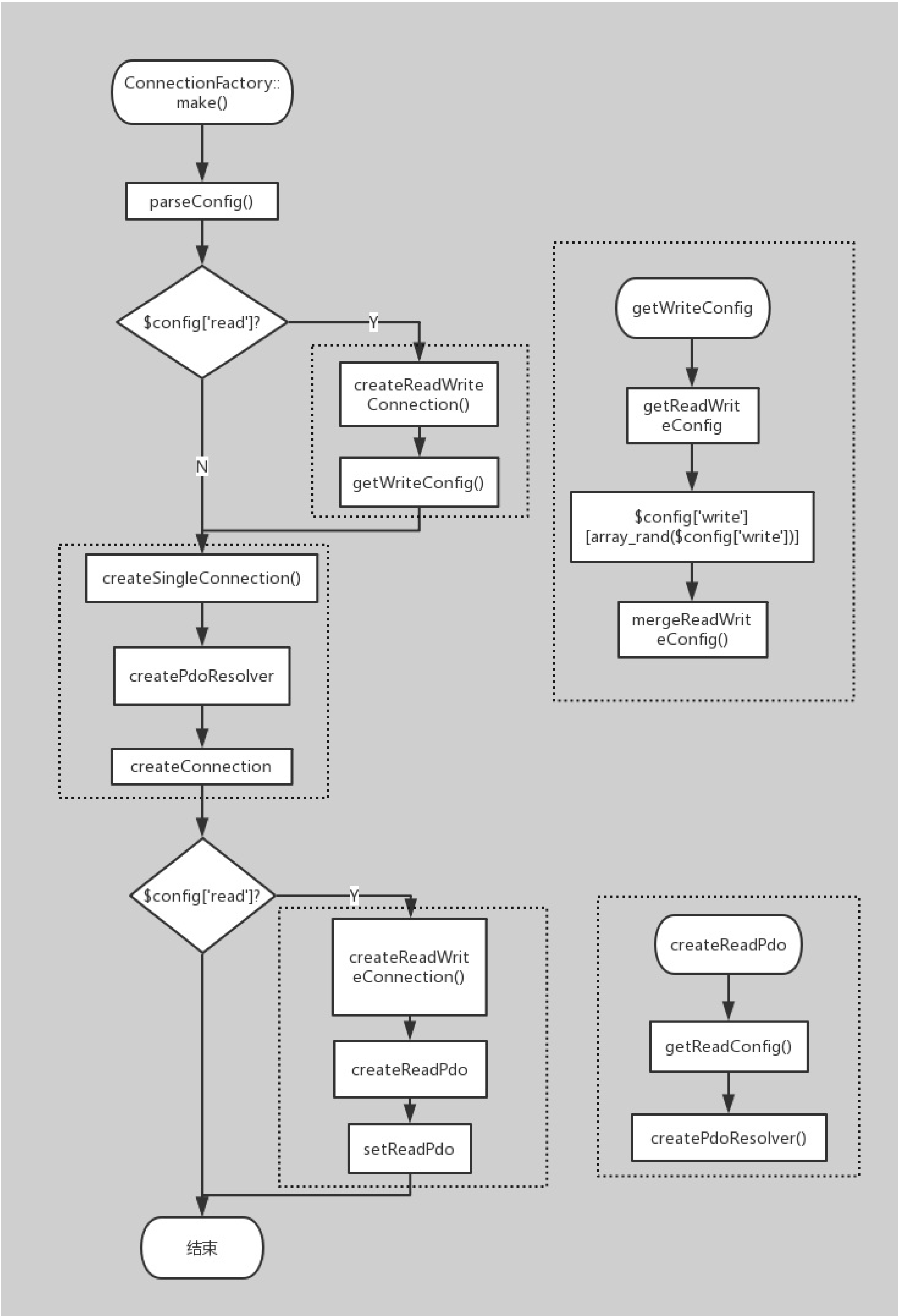
`setPdoForType` 函数就是根据 `type` 来设置读写：

当我们需要 `read` 数据库连接时，我们将 `read-pdo` 设置为主 `pdo` 。当我们需要 `write` 数据库连接时，我们将读写 `pdo` 都设置为 `write-pdo` ：

```
protected function setPdoForType(Connection $connection, $type =
 null)
{
    if ($type == 'read') {
        $connection->setPdo($connection->getReadPdo());
    } elseif ($type == 'write') {
        $connection->setReadPdo($connection->getPdo());
    }

    return $connection;
}
```

ConnectionFactory——数据库连接对象工厂


ConnectionFactory::
make()
parseConfig()
$config['read']?
Y
createReadWrite
Connection()
getWriteConfig()
N
getWriteConfig
getReadWrit
eConfig
$config['write']
[array_rand($config['write'])]
mergeReadWrit
eConfig()
createSingleConnection()
createPdoResolver
createConnection
$config['read']?
Y
createReadWrit
eConnection()
createReadPdo
setReadPdo
createReadPdo
getReadConfig()
createPdoResolver()
结束

make 函数——工厂接口

获取到了数据库的配置参数之后，就要利用 `ConnectionFactory` 来获取 `connection` 对象了：

```
public function make(array $config, $name = null)
{
    $config = $this->parseConfig($config, $name);

    if (isset($config['read'])) {
        return $this->createReadWriteConnection($config);
    }

    return $this->createSingleConnection($config);
}

protected function parseConfig(array $config, $name)
{
    return Arr::add(Arr::add($config, 'prefix', ''), 'name', $na
me);
}
```

在建立连接之前，要先向配置参数中添加默认的 `prefix` 属性与 `name` 属性。

接着，就要判断我们在配置文件中是否设置了读写分离。如果设置了读写分离，那么就会调用 `createReadWriteConnection` 函数，生成具有读、写两个功能的 `connection`；否则的话，就会调用 `createSingleConnection` 函数，生成普通的连接对象。

createSingleConnection 函数——制造数据库连接对象

`createSingleConnection` 函数是类 `ConnectionFactory` 的核心，用于生成新的数据库连接对象。

```
protected function createSingleConnection(array $config)
{
    $pdo = $this->createPdoResolver($config);

    return $this->createConnection(
        $config['driver'], $pdo, $config['database'], $config['p
refix'], $config
    );
}
```

`ConnectionFactory` 也很简单，只做了两件事情：制造 `pdo` 连接的闭包函数、构造一个新的 `connection` 对象。

createPdoResolver——数据库连接器闭包函数

根据配置参数中是否含有 `host`，创建不同的闭包函数：

```
protected function createPdoResolver(array $config)
{
    return array_key_exists('host', $config)
                        ? $this->createPdoResolverWithHosts($con
fig)
                        : $this->createPdoResolverWithoutHosts($
config);
}
```

不带有 `host` 的 `pdo` 闭包函数：

```
protected function createPdoResolverWithoutHosts(array $config)
{
    return function () use ($config) {
        return $this->createConnector($config)->connect($config)
;
    };
}
```

可以看出，不带有 `pdo` 的闭包函数非常简单，仅仅创建 `connector` 对象，利用 `connector` 对象进行数据库的连接。

带有 `host` 的 `pdo` 闭包函数：

```
protected function createPdoResolverWithHosts(array $config)
{
    return function () use ($config) {
        foreach (Arr::shuffle($hosts = $this->parseHosts($config
)) as $key => $host) {
            $config['host'] = $host;

            try {
                return $this->createConnector($config)->connect(
$config);
            } catch (PDOException $e) {
                if (count($hosts) - 1 === $key && $this->contain
er->bound(ExceptionHandler::class)) {
                    $this->container->make(ExceptionHandler::cla
ss)->report($e);
                }
            }
        }

        throw $e;
    };
}

protected function parseHosts(array $config)
{
    $hosts = array_wrap($config['host']);

    if (empty($hosts)) {
        throw new InvalidArgumentException('Database hosts array
 is empty.');
    }

    return $hosts;
}
```

带有 `host` 的闭包函数相对比较复杂，首先程序会随机选择不同的数据库依次来建立数据库连接，若均失败，就会报告异常。

createConnector——创建连接器

程序会根据配置参数中 `driver` 的不同来创建不同的连接器，每个连接器都继承自 `connector` 类，用于连接数据库。

```
public function createConnector(array $config)
{
    if (! isset($config['driver'])) {
        throw new InvalidArgumentException('A driver must be spe
cified.');
    }

    if ($this->container->bound($key = "db.connector.{$config['d
river']}")) {
        return $this->container->make($key);
    }

    switch ($config['driver']) {
        case 'mysql':
            return new MySqlConnector;
        case 'pgsql':
            return new PostgresConnector;
        case 'sqlite':
            return new SQLiteConnector;
        case 'sqlsrv':
            return new SqlServerConnector;
    }

    throw new InvalidArgumentException("Unsupported driver [{$co
nfig['driver']}]");
}
```

createConnection——创建连接对象

```
protected function createConnection($driver, $connection, $datab
ase, $prefix = '', array $config = [])
{
    if ($resolver = Connection::getResolver($driver)) {
        return $resolver($connection, $database, $prefix, $confi
g);
    }

    switch ($driver) {
        case 'mysql':
            return new MySqlConnection($connection, $database, $
prefix, $config);
        case 'pgsql':
            return new PostgresConnection($connection, $database
, $prefix, $config);
        case 'sqlite':
            return new SQLiteConnection($connection, $database,
$prefix, $config);
        case 'sqlsrv':
            return new SqlServerConnection($connection, $databas
e, $prefix, $config);
    }

    throw new InvalidArgumentException("Unsupported driver [$dri
ver]");
}
```

创建 `pdo` 闭包函数之后，会将该闭包函数放入 `connection` 对象当中去。以后我们利用 `connection` 对象进行查询或者更新数据库时，程序便会运行该闭包函数，与数据库进行连接。

createReadWriteConnection——创建读写连接对象

当配置文件中有 `read` 、 `write` 等配置项时，说明用户希望创建一个可以读写分离的数据库连接，此时：

```
protected function createReadWriteConnection(array $config)
{
    $connection = $this->createSingleConnection($this->getWriteC
onfig($config));

    return $connection->setReadPdo($this->createReadPdo($config)
);
}

protected function getWriteConfig(array $config)
{
    return $this->mergeReadWriteConfig(
        $config, $this->getReadWriteConfig($config, 'write')
    );
}

protected function getReadWriteConfig(array $config, $type)
{
    return isset($config[$type][0])
                    ? $config[$type][array_rand($config[$type])]
                    : $config[$type];
}

protected function mergeReadWriteConfig(array $config, array $me
rge)
{
    return Arr::except(array_merge($config, $merge), ['read', 'w
rite']);
}
```

可以看出，程序先读出关于 `write` 数据库的配置，之后将其合并到总配置当中，删除关于 `read` 数据库的配置，然后进行 `createSingleConnection` 建立新的连接对象。

建立连接对象之后，再根据 `read` 数据库的配置，生成 `read` 数据库的 `pdo` 闭包函数，并调用 `setReadPdo` 将其设置为读库 `pdo` 。

```
protected function createReadPdo(array $config)
{
    return $this->createPdoResolver($this->getReadConfig($config
));
}

protected function getReadConfig(array $config)
{
    return $this->mergeReadWriteConfig(
        $config, $this->getReadWriteConfig($config, 'read')
    );
}
```

connector 连接

我们以 `mysql` 为例：

```
class MySqlConnector extends Connector implements ConnectorInter
face
{
    public function connect(array $config)
    {
        $dsn = $this->getDsn($config);

        $options = $this->getOptions($config);

        $connection = $this->createConnection($dsn, $config, $op
tions);

            if (! empty($config['database'])) {
             $connection->exec("use `{$config['database']}`;");
        }

        $this->configureEncoding($connection, $config);

        $this->configureTimezone($connection, $config);

            $this->setModes($connection, $config);

      return $connection;
    }
}
```

getDsn——获取数据库连接DSN参数

```
protected function getDsn(array $config)
{
    return $this->hasSocket($config)
                        ? $this->getSocketDsn($config)
                        : $this->getHostDsn($config);
}

protected function hasSocket(array $config)
{
    return isset($config['unix_socket']) && ! empty($config['uni
x_socket']);
}

protected function getSocketDsn(array $config)
{
    return "mysql:unix_socket={$config['unix_socket']};dbname={$
config['database']}";
}

protected function getHostDsn(array $config)
{
    extract($config, EXTR_SKIP);

    return isset($port)
                ? "mysql:host={$host};port={$port};dbname={$data
base}"
                : "mysql:host={$host};dbname={$database}";
}
```

`mysql` 数据库的连接有两种：tcp连接与socket连接。

`socket` 连接更快，但是它要求应用程序与数据库在同一台机器，更普通的是使用 `tcp` 的方式连接数据库。框架根据配置参数来选择是采用 `socket` 还是 `tcp` 的方式连接数据库。

getOptions——pdo 属性设置

```
protected $options = [
    PDO::ATTR_CASE => PDO::CASE_NATURAL,
    PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
    PDO::ATTR_ORACLE_NULLS => PDO::NULL_NATURAL,
    PDO::ATTR_STRINGIFY_FETCHES => false,
    PDO::ATTR_EMULATE_PREPARES => false,
];

public function getOptions(array $config)
{
    $options = Arr::get($config, 'options', []);

    return array_diff_key($this->options, $options) + $options;
}
```

`pdo` 的属性主要有以下几种：

- `PDO::ATTR_CASE` 强制列名为指定的大小写。他的$value可为：
    - `PDO::CASE_LOWER` ：强制列名小写。
    - `PDO::CASE_NATURAL` ：保留数据库驱动返回的列名。
    - `PDO::CASE_UPPER` ：强制列名大写。
- `PDO::ATTR_ERRMODE` ：错误报告。他的$value可为：
    - `PDO::ERRMODE_SILENT` ： 仅设置错误代码。
    - `PDO::ERRMODE_WARNING` : 引发 E_WARNING 错误.
    - `PDO::ERRMODE_EXCEPTION` : 抛出 exceptions 异常。
- `PDO::ATTR_ORACLE_NULLS` （在所有驱动中都可用，不仅限于Oracle）： 转换 NULL 和空字符串。他的$value可为:
    - `PDO::NULL_NATURAL` : 不转换。
    - `PDO::NULL_EMPTY_STRING` ： 将空字符串转换成 NULL 。
    - `PDO::NULL_TO_STRING` : 将 NULL 转换成空字符串。
- `PDO::ATTR_STRINGIFY_FETCHES` : 提取的时候将数值转换为字符串。
- `PDO::ATTR_EMULATE_PREPARES` 启用或禁用预处理语句的模拟。 有些驱动不支持或有限度地支持本地预处理。使用此设置强制PDO总是模拟预处理语句（如果为 TRUE ），或试着使用本地预处理语句（如果为 `FALSE` ）。如果驱动不能成功预处理当前查询，它将总是回到模拟预处理语句上。 需要 `bool` 类型。
- `PDO::ATTR_AUTOCOMMIT` ：设置当前连接 `Mysql` 服务器的客户端的SQL语

句是否自动执行，默认是自动提交.

- `PDO::ATTR_PERSISTENT` ：当前对Mysql服务器的连接是否是长连接.

createConnection——创建数据库连接

```
public function createConnection($dsn, array $config, array $options)
{
    list($username, $password) = [
        Arr::get($config, 'username'), Arr::get($config, 'password'),
    ];

    try {
        return $this->createPdoConnection(
            $dsn, $username, $password, $options
        );
    } catch (Exception $e) {
        return $this->tryAgainIfCausedByLostConnection(
            $e, $dsn, $username, $password, $options
        );
    }
}

protected function createPdoConnection($dsn, $username, $password, $options)
{
    if (class_exists(PDOConnection::class) && ! $this->isPersistentConnection($options)) {
        return new PDOConnection($dsn, $username, $password, $options);
    }

    return new PDO($dsn, $username, $password, $options);
}
```

当 `pdo` 对象成功的建立起来后，说明我们已经与数据库成功地建立起来了一个连接，接下来我们就可以利用这个 `pdo` 对象进行查询或者更新等操作。

当创建 `pdo` 的时候抛出异常时：

```
protected function tryAgainIfCausedByLostConnection(Exception $e
, $dsn, $username, $password, $options)
{
    if ($this->causedByLostConnection($e)) {
        return $this->createPdoConnection($dsn, $username, $pass
word, $options);
    }

    throw $e;
}

protected function causedByLostConnection(Exception $e)
{
    $message = $e->getMessage();

    return Str::contains($message, [
        'server has gone away',
        'no connection to the server',
        'Lost connection',
        'is dead or not enabled',
        'Error while sending',
        'decryption failed or bad record mac',
        'server closed the connection unexpectedly',
        'SSL connection has been closed unexpectedly',
        'Error writing data to the connection',
        'Resource deadlock avoided',
    ]);
}
```

当判断出的异常是上面几种情况时，框架会再次尝试连接数据库。

configureEncoding——设置字符集与校对集

```
protected function configureEncoding($connection, array $config)
{
    if (! isset($config['charset'])) {
        return $connection;
    }

    $connection->prepare(
        "set names '{$config['charset']}'".$this->getCollation($
config)
    )->execute();
}

protected function getCollation(array $config)
{
    return ! is_null($config['collation']) ? " collate '{$config
['collation']}'" : '';
}
```

如果配置参数中设置了字符集与校对集，程序会利用配置的参数对数据库进行相关设置。

所谓的字符集与校对集设置，可以参考mysql 中 character set 与 collation 的点滴理解

configureTimezone——设置时间区

```
protected function configureTimezone($connection, array $config)
~
{
    if (isset($config['timezone'])) {
        $connection->prepare('set time_zone="'.$config['timezone'
].'"')->execute();
    }
}
```

setModes——设置 SQL_MODE 模式

```
protected function setModes(PDO $connection, array $config)
{
    if (isset($config['modes'])) {
        $this->setCustomModes($connection, $config);
    } elseif (isset($config['strict'])) {
        if ($config['strict']) {
            $connection->prepare($this->strictMode())->execute();
        } else {
            $connection->prepare("set session sql_mode='NO_ENGINE_SUBSTITUTION'")->execute();
        }
    }
}

protected function setCustomModes(PDO $connection, array $config)

{
    $modes = implode(',', $config['modes']);

    $connection->prepare("set session sql_mode='{$modes}'")->execute();
}

protected function strictMode()
{
    return "set session sql_mode='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION'";
}
```

以下内容参考：mysql的sql_mode设置简介:

`SQL_MODE` 直接理解就是：sql的运作模式。官方的说法是：sql_mode可以影响sql支持的语法以及数据的校验执行，这使得MySQL可以运行在不同的环境中以及和其他数据库一起运作。

想设置sql_mode有三种方式：

- 在命令行启动MySQL时添加参数 --sql-mode="modes"
- 在MySQL的配置文件（my.cnf或者my.ini）中添加一个配置sql-mode="modes"
- 运行时修改SQL mode可以通过以下命令之一：

```
SET GLOBAL sql_mode = 'modes';
SET SESSION sql_mode = 'modes';
```

几种常见的mode介绍

- `ONLY_FULL_GROUP_BY` ：出现在 `select` 语句、`HAVING` 条件和 `ORDER BY` 语句中的列，必须是 `GROUP BY` 的列或者依赖于 `GROUP BY` 列的函数列。

- `NO_AUTO_VALUE_ON_ZERO` ：该值影响自增长列的插入。默认设置下，插入0或 `NULL` 代表生成下一个自增长值。如果用户 希望插入的值为0，而该列又是自增长的，那么这个选项就有用了。

- `STRICT_TRANS_TABLES` ：在该模式下，如果一个值不能插入到一个事务表中，则中断当前的操作，对非事务表不做限制

- `NO_ZERO_IN_DATE` ：这个模式影响了是否允许日期中的月份和日包含0。如果开启此模式，2016-01-00是不允许的，但是0000-02-01是允许的。它实际的行为受到 `strict mode` 是否开启的影响1。

- `NO_ZERO_DATE` ：设置该值，`mysql` 数据库不允许插入零日期。它实际的行为受到 `strict mode` 是否开启的影响2。

- `ERROR_FOR_DIVISION_BY_ZERO` ：在 `INSERT` 或 `UPDATE` 过程中，如果数据被零除，则产生错误而非警告。如 果未给出该模式，那么数据被零除时 `MySQL` 返回 `NULL`

- `NO_AUTO_CREATE_USER` ：禁止 `GRANT` 创建密码为空的用户

- `NO_ENGINE_SUBSTITUTION` ：如果需要的存储引擎被禁用或未编译，那么抛出错误。不设置此值时，用默认的存储引擎替代，并抛出一个异常

- `PIPES_AS_CONCAT` ：将"||"视为字符串的连接操作符而非或运算符，这和Oracle数据库是一样的，也和字符串的拼接函数Concat相类似

- `ANSI_QUOTES` ：启用 `ANSI_QUOTES` 后，不能用双引号来引用字符串，因为它被解释为识别符

前言

当 `connection` 对象构建初始化完成后，我们就可以利用 `DB` 来进行数据库的 `CRUD` (`Create` 、 `Retrieve` 、 `Update` 、 `Delete`)操作。本篇文章，我们将会讲述 `laravel` 如何与 `pdo` 交互，实现基本数据库服务的原理。

run

`laravel` 中任何数据库的操作都要经过 `run` 这个函数，这个函数作用在于重新连接数据库、记录数据库日志、数据库异常处理：

```
protected function run($query, $bindings, Closure $callback)
{
    $this->reconnectIfMissingConnection();

    $start = microtime(true);

    try {
        $result = $this->runQueryCallback($query, $bindings, $ca
llback);
    } catch (QueryException $e) {
        $result = $this->handleQueryException(
            $e, $query, $bindings, $callback
        );
    }

    $this->logQuery(
        $query, $bindings, $this->getElapsedTime($start)
    );

    return $result;
}
```

重新连接数据库 reconnect

如果当期的 `pdo` 是空，那么就会调用 `reconnector` 重新与数据库进行连接：

```
protected function reconnectIfMissingConnection()
{
    if (is_null($this->pdo)) {
        $this->reconnect();
    }
}

public function reconnect()
{
    if (is_callable($this->reconnector)) {
        return call_user_func($this->reconnector, $this);
    }

    throw new LogicException('Lost connection and no reconnector
 available.');
}
```

运行数据库操作

数据库的 curd 操作会被包装成为一个闭包函数，作为 `runQueryCallback` 的一个参数，当运行正常时，会返回结果，如果遇到异常的话，会将异常转化为 `QueryException` ，并且抛出。

```
protected function runQueryCallback($query, $bindings, Closure $
callback)
{
    try {
        $result = $callback($query, $bindings);
    }

    catch (Exception $e) {
        throw new QueryException(
            $query, $this->prepareBindings($bindings), $e
        );
    }

    return $result;
}
```

数据库异常处理

当 `pdo` 查询返回异常的时候，如果当前是事务进行时，那么直接返回异常，让上一层事务来处理。

如果是由于与数据库事情连接导致的异常，那么就要重新与数据库进行连接：

```
protected function handleQueryException($e, $query, $bindings, C
losure $callback)
{
    if ($this->transactions >= 1) {
        throw $e;
    }

    return $this->tryAgainIfCausedByLostConnection(
        $e, $query, $bindings, $callback
    );
}
```

与数据库失去连接：

```
protected function tryAgainIfCausedByLostConnection(QueryExcepti
on $e, $query, $bindings, Closure $callback)
{
    if ($this->causedByLostConnection($e->getPrevious())) {
        $this->reconnect();

        return $this->runQueryCallback($query, $bindings, $callb
ack);
    }

    throw $e;
}

protected function causedByLostConnection(Exception $e)
{
    $message = $e->getMessage();

    return Str::contains($message, [
        'server has gone away',
        'no connection to the server',
        'Lost connection',
        'is dead or not enabled',
        'Error while sending',
        'decryption failed or bad record mac',
        'server closed the connection unexpectedly',
        'SSL connection has been closed unexpectedly',
        'Error writing data to the connection',
        'Resource deadlock avoided',
        'Transaction() on null',
        'child connection forced to terminate due to client_idle
_limit',
    ]);
}
```

数据库日志

```
public function logQuery($query, $bindings, $time = null)
{
    $this->event(new QueryExecuted($query, $bindings, $time, $th
is));

    if ($this->loggingQueries) {
        $this->queryLog[] = compact('query', 'bindings', 'time')
;
    }
}
```

想要开启或关闭日志功能：

```
public function enableQueryLog()
{
    $this->loggingQueries = true;
}

public function disableQueryLog()
{
    $this->loggingQueries = false;
}
```

Select 查询

```
public function select($query, $bindings = [], $useReadPdo = tru
e)
{
    return $this->run($query, $bindings, function ($query, $bind
ings) use ($useReadPdo) {
        if ($this->pretending()) {
            return [];
        }

        $statement = $this->prepared($this->getPdoForSelect($use
ReadPdo)
                          ->prepare($query));

        $this->bindValues($statement, $this->prepareBindings($bi
ndings));

        $statement->execute();

        return $statement->fetchAll();
    });
}
```

数据库的查询主要有一下几个步骤：

- 获取 `$this->pdo` 成员变量，若当前未连接数据库，则进行数据库连接，获取 `pdo` 对象。
- 设置 `pdo` 数据 `fetch` 模式
- `pdo` 进行 `sql` 语句预处理，`pdo` 绑定参数
- `sql` 语句执行，并获取数据。

getPdoForSelect 获取 pdo 对象

```
protected function getPdoForSelect($useReadPdo = true)
{
    return $useReadPdo ? $this->getReadPdo() : $this->getPdo();
}

public function getPdo()
{
    if ($this->pdo instanceof Closure) {
        return $this->pdo = call_user_func($this->pdo);
    }

    return $this->pdo;
}

public function getReadPdo()
{
    if ($this->transactions > 0) {
        return $this->getPdo();
    }

    if ($this->getConfig('sticky') && $this->recordsModified) {
        return $this->getPdo();
    }

    if ($this->readPdo instanceof Closure) {
        return $this->readPdo = call_user_func($this->readPdo);
    }

    return $this->readPdo ?: $this->getPdo();
}
```

`getPdo` 这里逻辑比较简单，值得我们注意的是 `getReadPdo` 。为了减缓数据库的压力，我们常常对数据库进行读写分离，也就是只要当写数据库这种操作发生时，才会使用写数据库，否则都会用读数据库。这种措施减少了数据库的压力，但是也带来了一些问题，那就是读写两个数据库在一定时间内会出现数据不一致的情况，原因就是写库的数据未能及时推送给读库，造成读库数据延迟的现象。为了在一定程度上解决这类问题， `laravel` 增添了 `sticky` 选项，

从程序中我们可以看出，当我们设置选项 `sticky` 为真，并且的确对数据库进行了写操作后，`getReadPdo` 会强制返回主库的连接，这样就避免了读写分离造成的延迟问题。

还有一种情况，当数据库在执行事务期间，所有的读取操作也会被强制连接主库。

prepared 设置数据获取方式

```
protected $fetchMode = PDO::FETCH_OBJ;
protected function prepared(PDOStatement $statement)
{
    $statement->setFetchMode($this->fetchMode);

    $this->event(new Events\StatementPrepared(
        $this, $statement
    ));

    return $statement;
}
```

`pdo` 的 `setFetchMode` 函数用于为语句设置默认的获取模式，通常模式有一下几种：

- PDO::FETCH_ASSOC //从结果集中获取以列名为索引的关联数组。
- PDO::FETCH_NUM //从结果集中获取一个以列在行中的数值偏移量为索引的值数组。
- PDO::FETCH_BOTH //这是默认值，包含上面两种数组。
- PDO::FETCH_OBJ //从结果集当前行的记录中获取其属性对应各个列名的一个对象。
- PDO::FETCH_BOUND //使用fetch()返回TRUE，并将获取的列值赋给在bindParm()方法中指定的相应变量。
- PDO::FETCH_LAZY //创建关联数组和索引数组，以及包含列属性的一个对象，从而可以在这三种接口中任选一种。

pdo 的 prepare 函数

`prepare` 函数会为 `PDOStatement::execute()` 方法准备要执行的 `SQL` 语句， `SQL` 语句可以包含零个或多个命名（ `:name` ）或问号（ `?` ）参数标记，参数在SQL执行时会被替换。

不能在 `SQL` 语句中同时包含命名（ `:name` ）或问号（ `?` ）参数标记，只能选择其中一种风格。

预处理 `SQL` 语句中的参数在使用 `PDOStatement::execute()` 方法时会传递真实的参数。

之所以使用 `prepare` 函数，是因为这个函数可以防止 `SQL` 注入，并且可以加快同一查询语句的速度。关于预处理与参数绑定防止 `SQL` 漏洞注入的原理可以参考：Web安全之SQL注入攻击技巧与防范.

pdo 的 bindValues 函数

在调用 `pdo` 的参数绑定函数之前， `laravel` 对参数值进一步进行了优化，把时间类型的对象利用 `grammer` 的设置重新格式化， `false` 也改为0。

`pdo` 的参数绑定函数 `bindValue` ，对于使用命名占位符的预处理语句，应是类似 :name 形式的参数名。对于使用问号占位符的预处理语句，应是以1开始索引的参数位置。

```
public function prepareBindings(array $bindings)
{
    $grammar = $this->getQueryGrammar();

    foreach ($bindings as $key => $value) {
        if ($value instanceof DateTimeInterface) {
            $bindings[$key] = $value->format($grammar->getDateFo
rmat());
        } elseif ($value === false) {
            $bindings[$key] = 0;
        }
    }

    return $bindings;
}

public function bindValues($statement, $bindings)
{
    foreach ($bindings as $key => $value) {
        $statement->bindValue(
            is_string($key) ? $key : $key + 1, $value,
            is_int($value) ? PDO::PARAM_INT : PDO::PARAM_STR
        );
    }
}
```

insert

```
public function insert($query, $bindings = [])
{
    return $this->statement($query, $bindings);
}

public function statement($query, $bindings = [])
{
    return $this->run($query, $bindings, function ($query, $bind
ings) {
        if ($this->pretending()) {
            return true;
        }

        $statement = $this->getPdo()->prepare($query);

        $this->bindValues($statement, $this->prepareBindings($bi
ndings));

        $this->recordsHaveBeenModified();

        return $statement->execute();
    });
}
```

这部分的代码与 `select` 非常相似，不同之处有一下几个：

- 直接获取写库的连接，不会考虑读库
- 由于不需要返回任何数据库数据，因此也不必设置 `fetchMode` 。
- `recordsHaveBeenModified` 函数标志当前连接数据库已被写入。
- 不需要调用函数 `fetchAll`

```
public function recordsHaveBeenModified($value = true)
{
    if (! $this->recordsModified) {
        $this->recordsModified = $value;
    }
}
```

update、delete

`affectingStatement` 这个函数与上面的 `statement` 函数一致，只是最后会返回更新、删除影响的行数。

```
public function update($query, $bindings = [])
{
    return $this->affectingStatement($query, $bindings);
}

public function delete($query, $bindings = [])
{
    return $this->affectingStatement($query, $bindings);
}

public function affectingStatement($query, $bindings = [])
{
    return $this->run($query, $bindings, function ($query, $bind
ings) {
        if ($this->pretending()) {
            return 0;
        }

        $statement = $this->getPdo()->prepare($query);

        $this->bindValues($statement, $this->prepareBindings($bi
ndings));

        $statement->execute();

        $this->recordsHaveBeenModified(
            ($count = $statement->rowCount()) > 0
        );

        return $count;
    });
}
```

transaction 数据库事务

为保持数据的一致性，对于重要的数据我们经常使用数据库事务，`transaction` 函数接受一个闭包函数，与一个重复尝试的次数：

```
public function transaction(Closure $callback, $attempts = 1)
{
    for ($currentAttempt = 1; $currentAttempt <= $attempts; $cur
rentAttempt++) {
        $this->beginTransaction();

        try {
            return tap($callback($this), function ($result) {
                $this->commit();
            });
        }

        catch (Exception $e) {
            $this->handleTransactionException(
                $e, $currentAttempt, $attempts
            );
        } catch (Throwable $e) {
            $this->rollBack();

            throw $e;
        }
    }
}
```

开始事务

数据库事务中非常重要的成员变量是 `$this->transactions` ，它标志着当前事务的进程：

```
public function beginTransaction()
{
    $this->createTransaction();

    ++$this->transactions;

    $this->fireConnectionEvent('beganTransaction');
}
```

可以看出，当创建事务成功后，就会累加 `$this->transactions` ，并且启动 `event` ，创建事务：

```
protected function createTransaction()
{
    if ($this->transactions == 0) {
        try {
            $this->getPdo()->beginTransaction();
        } catch (Exception $e) {
            $this->handleBeginTransactionException($e);
        }
    } elseif ($this->transactions >= 1 && $this->queryGrammar->s
upportsSavepoints()) {
        $this->createSavepoint();
    }
}
```

如果当前没有任何事务，那么就会调用 `pdo` 来开启事务。

如果当前已经在事务保护的范围内，那么就会创建 `SAVEPOINT` ，实现数据库嵌套事务：

```
protected function createSavepoint()
{
    $this->getPdo()->exec(
        $this->queryGrammar->compileSavepoint('trans'.($this->tr
ansactions + 1))
    );
}

public function compileSavepoint($name)
{
    return 'SAVEPOINT '.$name;
}
```

如果创建事务失败，那么就会调用 `handleBeginTransactionException` ：

```
protected function handleBeginTransactionException($e)
{
    if ($this->causedByLostConnection($e)) {
        $this->reconnect();

        $this->pdo->beginTransaction();
    } else {
        throw $e;
    }
}
```

如果创建事务失败是由于与数据库失去连接的话，那么就会重新连接数据库，否则就要抛出异常。

事务异常

事务的异常处理比较复杂，可以先看一看代码：

```
protected function handleTransactionException($e, $currentAttemp
t, $maxAttempts)
{
    if ($this->causedByDeadlock($e) &&
        $this->transactions > 1) {
        --$this->transactions;

        throw $e;
    }

    $this->rollBack();

    if ($this->causedByDeadlock($e) &&
        $currentAttempt < $maxAttempts) {
        return;
    }

    throw $e;
}

protected function causedByDeadlock(Exception $e)
{
    $message = $e->getMessage();

    return Str::contains($message, [
        'Deadlock found when trying to get lock',
        'deadlock detected',
        'The database file is locked',
        'database is locked',
        'database table is locked',
        'A table in the database is locked',
        'has been chosen as the deadlock victim',
        'Lock wait timeout exceeded; try restarting transaction'
,
    ]);
}
```

这里可以分为四种情况：

- 单一事务，非死锁导致的异常

单一事务就是说，此时的事务只有一层，没有嵌套事务的存在。数据库的异常也不是死锁导致的，一般是由于 `sql` 语句不正确引起的。这个时候，`handleTransactionException` 会直接回滚事务，并且抛出异常到外层：

```
try {
        return tap($callback($this), function ($result) {
            $this->commit();
        });
}
catch (Exception $e) {
     $this->handleTransactionException(
         $e, $currentAttempt, $attempts
     );
} catch (Throwable $e) {
       $this->rollBack();

       throw $e;
}
```

接到异常之后，程序会再次回滚，但是由于 `$this->transactions` 已经为 0，因此回滚直接返回，并未真正执行，之后就会抛出异常。

- 单一事务，死锁异常

有死锁导致的单一事务异常，一般是由于其他程序同时更改了数据库，这个时候，就要判断当前重复尝试的次数是否大于用户设置的 `maxAttempts` ，如果小于就继续尝试，如果大于，那么就会抛出异常。

- 嵌套事务，非死锁异常

如果出现嵌套事务，例如：

```
\DB::transaction(function(){
    ...
    //directly or indirectly call another transaction:
    \DB::transaction(function() {
        ...
        ...
    }, 2);//attempt twice
}, 2);//attempt twice
```

如果是非死锁导致的异常，那么就要首先回滚内层的事务，抛出异常到外层事务，再回滚外层事务，抛出异常，让用户来处理。也就是说，对于嵌套事务来说，内部事务异常，一定要回滚整个事务，而不是仅仅回滚内部事务。

- 嵌套事务，死锁异常

嵌套事务的死锁异常，仍然和嵌套事务非死锁异常一样，内部事务异常，一定要回滚整个事务。

但是，不同的是，`mysql` 对于嵌套事务的回滚会导致外部事务一并回滚:InnoDB Error Handling，因此这时，我们仅仅将 `$this->transactions` 减一，并抛出异常，使得外层事务回滚抛出异常即可。

回滚事务

如果事务内的数据库更新操作失败，那么就要进行回滚：

```
public function rollBack($toLevel = null)
{
    $toLevel = is_null($toLevel)
                ? $this->transactions - 1
                : $toLevel;

    if ($toLevel < 0 || $toLevel >= $this->transactions) {
        return;
    }

    $this->performRollBack($toLevel);

    $this->transactions = $toLevel;

    $this->fireConnectionEvent('rollingBack');
}
```

回滚的第一件事就是要减少 `$this->transactions` 的值，标志当前事务失败。

回滚的时候仍然要判断当前事务的状态，如果当前处于嵌套事务的话，就要进行回滚到 `SAVEPOINT`，如果是单一事务的话，才会真正回滚退出事务：

```
protected function performRollBack($toLevel)
{
    if ($toLevel == 0) {
        $this->getPdo()->rollBack();
    } elseif ($this->queryGrammar->supportsSavepoints()) {
        $this->getPdo()->exec(
            $this->queryGrammar->compileSavepointRollBack('trans'
.($toLevel + 1))
        );
    }
}

public function compileSavepointRollBack($name)
{
    return 'ROLLBACK TO SAVEPOINT '.$name;
}
```

提交事务

提交事务比较简单，仅仅是调用 `pdo` 的 `commit` 即可。需要注意的是对于嵌套事务的事务提交，`commit` 函数仅仅更新了 `$this->transactions` ，而并没有真正提交事务，原因是内层事务的提交对于 `mysql` 来说是无效的，只有外部事务的提交才能更新整个事务。

```
public function commit()
{
    if ($this->transactions == 1) {
        $this->getPdo()->commit();
    }

    $this->transactions = max(0, $this->transactions - 1);

    $this->fireConnectionEvent('committed');
}
```

paginate 分页

`laravel` 的分页用起来非常简单，只需要对 `query` 调用 `paginate` 函数，把返回的对象扔给前端 `blade` 文件，在 `blade` 文件调用函数 `render` 函数或者 `link` 函数，就可以得到 `上一页` 、 `下一页` 等等分页特效。

实际上，我们可以简单地把分页服务看作一个前端资源， `render` 函数或者 `link` 函数的结果就是分页前端代码。

如果你还对 `laravel` 的分页不是很熟悉，请先阅读官方文档 ： 分页。

分页服务的启动

分页功能也是由一个服务提供者所启动的， `PaginationServiceProvider` 就是负责注册和启动分页服务的服务提供者：

```
class PaginationServiceProvider extends ServiceProvider
{
    public function register()
    {
        Paginator::viewFactoryResolver(function () {
            return $this->app['view'];
        });

        Paginator::currentPathResolver(function () {
            return $this->app['request']->url();
        });

        Paginator::currentPageResolver(function ($pageName = 'pa
ge') {
            $page = $this->app['request']->input($pageName);

            if (filter_var($page, FILTER_VALIDATE_INT) !== false
 && (int) $page >= 1) {
                return $page;
            }

            return 1;
        });
    }
}
```

我们看到，服务提供者的注册函数为 `Paginator` 设置三个闭包函数：

- viewFactoryResolver 为 `Paginator` 设置了生成前端资源的类，用于获取分页前端代码。
- currentPathResolver 为 `Paginator` 设置了 `url` 的地址。我们知道，`上一页`、`下一页` 等等都是可以执行翻页的操作，所以实际上这些按钮必然含有链接，而链接的地址就是当前请求的 `url` 地址，不同的按钮的链接地址只是 `page` 的参数不同而已。
- currentPageResolver 为 `Paginator` 获取了当前的页数。

```
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/resources/views', 'pagination'
);

    if ($this->app->runningInConsole()) {
        $this->publishes([
            __DIR__.'/resources/views' => $this->app->resourcePa
th('views/vendor/pagination'),
        ], 'laravel-pagination');
    }
}

protected function loadViewsFrom($path, $namespace)
{
    if (is_dir($appPath = $this->app->resourcePath().'/views/ven
dor/'.$namespace)) {
        $this->app['view']->addNamespace($namespace, $appPath);
    }

    $this->app['view']->addNamespace($namespace, $path);
}
```

服务的启动函数为分页服务设置了默认的前端分页资源。

分页服务 paginator

分页服务 `paginator` 函数用于 `queryBuilder`，用于获取分页的数据库数据：

```
public function paginate($perPage = 15, $columns = ['*'], $pageN
ame = 'page', $page = null)
{
    $page = $page ?: Paginator::resolveCurrentPage($pageName);

    $total = $this->getCountForPagination($columns);

    $results = $total
        ? $this->forPage($page, $perPage)->get($columns) : colle
ct();

    return $this->paginator($results, $total, $perPage, $page, [
        'path' => Paginator::resolveCurrentPath(),
        'pageName' => $pageName,
    ]);
}

protected function paginator($items, $total, $perPage, $currentP
age, $options)
{
    return Container::getInstance()->makeWith(LengthAwarePaginat
or::class, compact(
        'items', 'total', 'perPage', 'currentPage', 'options'
    ));
}
```

也就是说，当我们写下这样的代码时：

```
DB::table('user')->select('*')->where('status',1)->paginator();
```

我们可以获取到一个 `LengthAwarePaginator` 类对象，对这个对象调用 `render` 函数就可以获取分页前端资源。

我们先来研究一下 `paginator` 函数。

获取当前页

我们可以看到，在这个函数中程序先获取当前页数：

```
public static function resolveCurrentPage($pageName = 'page', $default = 1)
{
    if (isset(static::$currentPageResolver)) {
        return call_user_func(static::$currentPageResolver, $pageName);
    }

    return $default;
}
```

currentPageResolver 就是上一节中 `currentPageResolver` 设置的闭包函数，这个闭包函数从请求参数中获取当前页：

```
$page = $this->app['request']->input($pageName);
```

获取数据库总记录数

计算数据库符合搜索条件的总记录数理所当然的是使用聚合函数 `count` ：

```
public function getCountForPagination($columns = ['*'])
{
    $results = $this->runPaginationCountQuery($columns);

    if (isset($this->groups)) {
        return count($results);
    } elseif (! isset($results[0])) {
        return 0;
    } elseif (is_object($results[0])) {
        return (int) $results[0]->aggregate;
    } else {
        return (int) array_change_key_case((array) $results[0])[
'aggregate'];
    }
}

protected function runPaginationCountQuery($columns = ['*'])
{
    return $this->cloneWithout(['columns', 'orders', 'limit', 'o
ffset'])
                ->cloneWithoutBindings(['select', 'order'])
                ->setAggregate('count', $this->withoutSelectAlia
ses($columns))
                ->get()->all();
}
```

获取当前页数据

获取当前页当然是使用 `forPage` 函数：

```
$results = $total
        ? $this->forPage($page, $perPage)->get($columns) : colle
ct();
```

初始化 LengthAwarePaginator

`paginator` 函数利用 loc 容器来生成 `LengthAwarePaginator` 实例：

```
protected function paginator($items, $total, $perPage, $currentP
age, $options)
{
    return Container::getInstance()->makeWith(LengthAwarePaginat
or::class, compact(
        'items', 'total', 'perPage', 'currentPage', 'options'
    ));
}
```

`LengthAwarePaginator` 的初始化：

```
public function __construct($items, $total, $perPage, $currentPa
ge = null, array $options = [])
{
    foreach ($options as $key => $value) {
        $this->{$key} = $value;
    }

    $this->total = $total;
    $this->perPage = $perPage;
    $this->lastPage = max((int) ceil($total / $perPage), 1);
    $this->path = $this->path !== '/' ? rtrim($this->path, '/')
: $this->path;
    $this->currentPage = $this->setCurrentPage($currentPage, $th
is->pageName);
    $this->items = $items instanceof Collection ? $items : Colle
ction::make($items);
}
```

分页资源 render

对 `LengthAwarePaginator` 调用 `render` 函数会得到分页所需要的前端资源：

```
public function render($view = null, $data = [])
{
    return new HtmlString(static::viewFactory()->make($view ?: static::$defaultView, array_merge($data, [
        'paginator' => $this,
        'elements' => $this->elements(),
    ]))->render());
}
```

当我们使用默认的分页样式的时候，不需要向 `render` 函数传入 `view` 参数，此时程序会自动加载默认的前端资源:

```
public static $defaultView = 'pagination::default';
```

该资源的默认地址是 `illuminate\Pagination\resources\views\default.blade.php` :

```
@if ($paginator->hasPages())
    <ul class="pagination">
        {{-- Previous Page Link --}}
        @if ($paginator->onFirstPage())
            <li class="disabled"><span><<</span></li>
        @else
            <li><a href="{{ $paginator->previousPageUrl() }}" rel="prev"><<</a></li>
        @endif

        {{-- Pagination Elements --}}
        @foreach ($elements as $element)
            {{-- "Three Dots" Separator --}}
            @if (is_string($element))
                <li class="disabled"><span>{{ $element }}</span></li>
            @endif

            {{-- Array Of Links --}}
            @if (is_array($element))
                @foreach ($element as $page => $url)
```

```
                    @if ($page == $paginator->currentPage())
                        <li class="active"><span>{{ $page }}</sp
an></li>
                    @else
                        <li><a href="{{ $url }}">{{ $page }}</a>
</li>
                    @endif
                @endforeach
            @endif
        @endforeach

        {{-- Next Page Link --}}
        @if ($paginator->hasMorePages())
            <li><a href="{{ $paginator->nextPageUrl() }}" rel="n
ext">>></a></li>
        @else
            <li class="disabled"><span>>></span></li>
        @endif
    </ul>
@endif
```

可以看到，分页效果的代码分为三部分：前一页、后一页、分页元素。

前一页

如果当前页是第一页的话，`前一页` 按钮需要置灰：

```
public function onFirstPage()
{
    return $this->currentPage() <= 1;
}
```

否则的话，就要为 `前一页` 按钮赋予链接：

```
public function previousPageUrl()
{
    if ($this->currentPage() > 1) {
        return $this->url($this->currentPage() - 1);
    }
}

public function url($page)
{
    if ($page <= 0) {
        $page = 1;
    }

    $parameters = [$this->pageName => $page];

    if (count($this->query) > 0) {
        $parameters = array_merge($this->query, $parameters);
    }

    return $this->path
                    .(Str::contains($this->path, '?') ? '&' : '?'
)
                    .http_build_query($parameters, '', '&')
                    .$this->buildFragment();
}
```

如果列表页中存在一些搜索条件，这些搜索条件会被加载到 `$this->query` 成员变量中，生成 `url` 的时候，这些搜索添加会被加到 `request` 的参数中。可以使用 `append` 方法附加查询参数到分页链接中：

```
public function appends($key, $value = null)
{
    if (is_array($key)) {
        return $this->appendArray($key);
    }

    return $this->addQuery($key, $value);
}

protected function appendArray(array $keys)
{
    foreach ($keys as $key => $value) {
        $this->addQuery($key, $value);
    }

    return $this;
}
```

下一页

与 `前一页` 类似，如果已经在最后一页，那么 `下一页` 按钮将会被置灰：

```
public function hasMorePages()
{
    return $this->currentPage() < $this->lastPage();
}
```

下一页的链接：

```
public function nextPageUrl()
{
    if ($this->lastPage() > $this->currentPage()) {
        return $this->url($this->currentPage() + 1);
    }
}
```

`上一页` 与 `下一页` 按钮的功能比较简单，至于中间的分页特效比较复杂，我们由下一节来说。

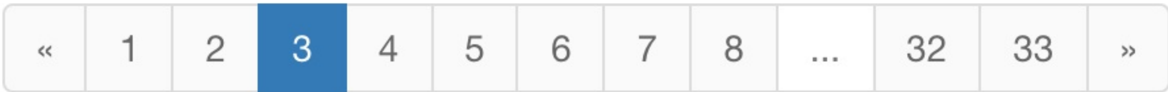
分页 elements

我们先说一下不同的分页样式：

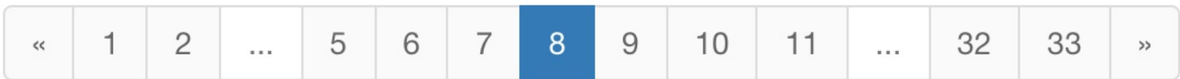
- 当我们设置两侧页数为 3 时，当前数据总页数小于 8 页时分页效果：



- 总页数大于 6 页，且当前页在前 8 页（2 * 3 + 2）时分页效果：



- 当前页在前 6 页与后 6 页之间分页效果：



- 当前页在最后 6 页时分页效果：



分页效果样式的关键来源于 `UrlWindow` ，这个类用于根据总页数与当前页的不同来控制不同的分页样式。

```
protected function elements()
{
    $window = UrlWindow::make($this);

    return array_filter([
        $window['first'],
        is_array($window['slider']) ? '...' : null,
        $window['slider'],
        is_array($window['last']) ? '...' : null,
        $window['last'],
    ]);
}

public static function make(PaginatorContract $paginator, $onEac
hSide = 3)
{
    return (new static($paginator))->get($onEachSide);
}

public function get($onEachSide = 3)
{
    if ($this->paginator->lastPage() < ($onEachSide * 2) + 6) {
        return $this->getSmallSlider();
    }

    return $this->getUrlSlider($onEachSide);
}
```

小型分页 getSmallSlider

如果当前总页数小于 `($onEachSide * 2) + 6` 的话，就会调用小型分页效果，这种小型分页效果直接将所有页数全部显示：

```
protected function getSmallSlider()
{
    return [
        'first'  => $this->paginator->getUrlRange(1, $this->last
Page()),
        'slider' => null,
        'last'   => null,
    ];
}

public function getUrlRange($start, $end)
{
    return collect(range($start, $end))->mapWithKeys(function ($
page) {
        return [$page => $this->url($page)];
    })->all();
}
```

CloseToBeginning 分页效果

当前页数位于前 `($onEachSide * 2)` 页时：

```
protected function getUrlSlider($onEachSide)
{
    $window = $onEachSide * 2;

    if (! $this->hasPages()) {
        return ['first' => null, 'slider' => null, 'last' => null
];
    }

    if ($this->currentPage() <= $window) {
        return $this->getSliderTooCloseToBeginning($window);
    }

    elseif ($this->currentPage() > ($this->lastPage() - $window)
) {
        return $this->getSliderTooCloseToEnding($window);
    }

    return $this->getFullSlider($onEachSide);
}

protected function getSliderTooCloseToBeginning($window)
{
    return [
        'first' => $this->paginator->getUrlRange(1, $window + 2)
,
        'slider' => null,
        'last' => $this->getFinish(),
    ];
}

public function getFinish()
{
    return $this->paginator->getUrlRange(
        $this->lastPage() - 1,
        $this->lastPage()
    );
}
```

假设我们设置当前两侧页数为 3，当前页为 5，总页数22，函数 `getSliderTooCloseToBeginning` 返回结果为：

```
return [
    'first' => [
        1 => '/www.example.com/example?page=1',
        2 => '/www.example.com/example?page=2'
        3 => '/www.example.com/example?page=3'
        4 => '/www.example.com/example?page=4'
        5 => '/www.example.com/example?page=5'
        6 => '/www.example.com/example?page=6'
        7 => '/www.example.com/example?page=7'
        8 => '/www.example.com/example?page=8'],
    'slider' => null,
    'last' => [
        21 => '/www.example.com/example?page=21',
        22 => '/www.example.com/example?page=22'],
];
```

这个时候 `element` 函数返回数据：

```
protected function elements()
{
    $window = UrlWindow::make($this);

    return array_filter([
        $window['first'],
        is_array($window['slider']) ? '...' : null,
        $window['slider'],
        is_array($window['last']) ? '...' : null,
        $window['last'],
    ]);
}

//返回结果
[
    [
        1 => '/www.example.com/example?page=1',
        2 => '/www.example.com/example?page=2',
        3 => '/www.example.com/example?page=3',
        4 => '/www.example.com/example?page=4',
        5 => '/www.example.com/example?page=5',
        6 => '/www.example.com/example?page=6',
        7 => '/www.example.com/example?page=7',
        8 => '/www.example.com/example?page=8',
    ],                                              //$window['first']
    '...',                                          //is_array($window
['last']) ? '...' : null
    [
        21 => '/www.example.com/example?page=21',
        22 => '/www.example.com/example?page=22',
    ],                                              //$window['last']
]
```

TooCloseToEnding 分页效果

当前页数位于后 `($onEachSide * 2)` 页时：

```
protected function getSliderTooCloseToEnding($window)
{
    $last = $this->paginator->getUrlRange(
        $this->lastPage() - ($window + 2),
        $this->lastPage()
    );

    return [
        'first' => $this->getStart(),
        'slider' => null,
        'last' => $last,
    ];
}

public function getStart()
{
    return $this->paginator->getUrlRange(1, 2);
}
```

假设我们设置当前两侧页数为 3，当前页为 18，总页数22，函数 `getSliderTooCloseToEnding` 返回结果为：

```
return [
    'first' => [
        1 => '/www.example.com/example?page=1',
        2 => '/www.example.com/example?page=2'
    ],
    'slider' => null,
    'last' => [
        15 => '/www.example.com/example?page=15',
        16 => '/www.example.com/example?page=16',
        17 => '/www.example.com/example?page=17',
        18 => '/www.example.com/example?page=18',
        19 => '/www.example.com/example?page=19',
        20 => '/www.example.com/example?page=20',
        21 => '/www.example.com/example?page=21',
        22 => '/www.example.com/example?page=22',
    ],
];
```

这个时候 `element` 函数返回数据：

```
[
    [
        1 => '/www.example.com/example?page=1',
        2 => '/www.example.com/example?page=2'
   ],
   '...',
   [
           15 => '/www.example.com/example?page=15',
           16 => '/www.example.com/example?page=16',
           17 => '/www.example.com/example?page=17',
           18 => '/www.example.com/example?page=18',
           19 => '/www.example.com/example?page=19',
           20 => '/www.example.com/example?page=20',
           21 => '/www.example.com/example?page=21',
           22 => '/www.example.com/example?page=22',
   ]
]
```

FullSlider 分页效果

当前页数位于中间时：

```
protected function getFullSlider($onEachSide)
{
    return [
        'first'  => $this->getStart(),
        'slider' => $this->getAdjacentUrlRange($onEachSide),
        'last'   => $this->getFinish(),
    ];
}

public function getAdjacentUrlRange($onEachSide)
{
    return $this->paginator->getUrlRange(
        $this->currentPage() - $onEachSide,
        $this->currentPage() + $onEachSide
    );
}
```

假设我们设置当前两侧页数为 3，当前页为 10，总页数22，函数 `getFullSlider` 返回结果为：

```php
return [
    'first' => [
        1 => '/www.example.com/example?page=1',
        2 => '/www.example.com/example?page=2'
    ],
    'slider' => [
        7 => '/www.example.com/example?page=7',
        8 => '/www.example.com/example?page=8',
        9 => '/www.example.com/example?page=9',
        10 => '/www.example.com/example?page=10',
        11 => '/www.example.com/example?page=11',
        12 => '/www.example.com/example?page=12',
        13 => '/www.example.com/example?page=13',
    ],
    'last' => [
        21 => '/www.example.com/example?page=21',
        22 => '/www.example.com/example?page=22',
    ],
];
```

这个时候 `element` 函数返回数据：

```
[
    [
        1 => '/www.example.com/example?page=1',
        2 => '/www.example.com/example?page=2'
   ],
   '...',
   [
       7 => '/www.example.com/example?page=7',
       8 => '/www.example.com/example?page=8',
       9 => '/www.example.com/example?page=9',
       10 => '/www.example.com/example?page=10',
       11 => '/www.example.com/example?page=11',
       12 => '/www.example.com/example?page=12',
       13 => '/www.example.com/example?page=13',
   ],
   '...',
   [
           21 => '/www.example.com/example?page=21',
           22 => '/www.example.com/example?page=22',
   ]
]
```

simplePaginate 简单分页

简单分页相比以上的功能来说，精简了 `elements` 的特效：

```
public function simplePaginate($perPage = 15, $columns = ['*'],
$pageName = 'page', $page = null)
{
    $page = $page ?: Paginator::resolveCurrentPage($pageName);

    $this->skip(($page - 1) * $perPage)->take($perPage + 1);

    return $this->simplePaginator($this->get($columns), $perPage
, $page, [
        'path' => Paginator::resolveCurrentPath(),
        'pageName' => $pageName,
    ]);
}

protected function simplePaginator($items, $perPage, $currentPag
e, $options)
{
    return Container::getInstance()->makeWith(Paginator::class,
compact(
        'items', 'perPage', 'currentPage', 'options'
    ));
}
```

分页服务的类不再使用 `LengthAwarePaginator` 类，而开始使用 `Paginator`，这两个类最大的不同在于 `render` 函数：

```
public static $defaultSimpleView = 'pagination::simple-default';

public function render($view = null, $data = [])
{
    return new HtmlString(
        static::viewFactory()->make($view ?: static::$defaultSim
pleView, array_merge($data, [
            'paginator' => $this,
        ]))->render()
    );
}
```

`render` 函数调用的前端资源默认地址为 `illuminate\Pagination\resources\views\simple-default.blade.php`:

```
@if ($paginator->hasPages())
    <ul class="pagination">
        {{-- Previous Page Link --}}
        @if ($paginator->onFirstPage())
            <li class="disabled"><span>@lang('pagination.previou
s')</span></li>
        @else
            <li><a href="{{ $paginator->previousPageUrl() }}" re
l="prev">@lang('pagination.previous')</a></li>
        @endif

        {{-- Next Page Link --}}
        @if ($paginator->hasMorePages())
            <li><a href="{{ $paginator->nextPageUrl() }}" rel="n
ext">@lang('pagination.next')</a></li>
        @else
            <li class="disabled"><span>@lang('pagination.next')<
/span></li>
        @endif
    </ul>
@endif
```

可以看到，简单分页只有 `上一页` 、 `下一页` 两个按钮。

获取模型

get 函数

```
public function get($columns = ['*'])
{
    $builder = $this->applyScopes();

    if (count($models = $builder->getModels($columns)) > 0) {
        $models = $builder->eagerLoadRelations($models);
    }

    return $builder->getModel()->newCollection($models);
}

public function getModels($columns = ['*'])
{
    return $this->model->hydrate(
        $this->query->get($columns)->all()
    )->all();
}
```

`get` 函数会将 `QueryBuilder` 所获取的数据进一步包装 `hydrate` 。`hydrate` 函数会将数据库取回来的数据打包成数据库模型对象 `Eloquent Model` ，如果可以获取到数据，还会利用函数 `eagerLoadRelations` 来预加载关系模型。

```
public function hydrate(array $items)
{
    $instance = $this->newModelInstance();

    return $instance->newCollection(array_map(function ($item) u
se ($instance) {
        return $instance->newFromBuilder($item);
    }, $items));
}
```

`newModelInstance` 函数创建了一个新的数据库模型对象，重要的是这个函数为新的数据库模型对象赋予了 `connection` ：

```
public function newModelInstance($attributes = [])
{
    return $this->model->newInstance($attributes)->setConnection
(
        $this->query->getConnection()->getName()
    );
}
```

`newFromBuilder` 函数会将所有数据库数据存入另一个新的 `Eloquent Model` 的 `attributes` 中：

```
public function newFromBuilder($attributes = [], $connection = n
ull)
{
    $model = $this->newInstance([], true);

    $model->setRawAttributes((array) $attributes, true);

    $model->setConnection($connection ?: $this->getConnectionNam
e());

    $model->fireModelEvent('retrieved', false);

    return $model;
}
```

newInstance 函数专用于创建新的数据库对象模型：

```
public function newInstance($attributes = [], $exists = false)
{
    $model = new static((array) $attributes);

    $model->exists = $exists;

    $model->setConnection(
        $this->getConnectionName()
    );

    return $model;
}
```

值得注意的是 `newInstance` 将 `exist` 设置为 `true` ，意味着当前这个数据库模型对象是从数据库中获取而来，并非是手动新建的，这个 `exist` 为真，我们才能对这个数据库对象进行 `update` 。

`setRawAttributes` 函数为新的数据库对象赋予属性值，并且进行 `sync` ，标志着对象的原始状态：

```
public function setRawAttributes(array $attributes, $sync = fals
e)
{
    $this->attributes = $attributes;

    if ($sync) {
        $this->syncOriginal();
    }

    return $this;
}

public function syncOriginal()
{
    $this->original = $this->attributes;

    return $this;
}
```

这个原始状态的记录十分重要，原因是 `save` 函数就是利用原始值 `original` 与属性值 `attributes` 的差异来决定更新的字段。

find 函数

`find` 函数用于利用主键 `id` 来查询数据，`find` 函数也可以传入数组，查询多个数据

```
public function find($id, $columns = ['*'])
{
    if (is_array($id) || $id instanceof Arrayable) {
        return $this->findMany($id, $columns);
    }

    return $this->whereKey($id)->first($columns);
}

public function findMany($ids, $columns = ['*'])
{
    if (empty($ids)) {
        return $this->model->newCollection();
    }

    return $this->whereKey($ids)->get($columns);
}
```

findOrFail

`laravel` 还提供 `findOrFail` 函数，一般用于 `controller`，在未找到记录的时候会抛出异常。

```
public function findOrFail($id, $columns = ['*'])
{
    $result = $this->find($id, $columns);

    if (is_array($id)) {
        if (count($result) == count(array_unique($id))) {
            return $result;
        }
    } elseif (! is_null($result)) {
        return $result;
    }

    throw (new ModelNotFoundException)->setModel(
        get_class($this->model), $id
    );
}
```

其他查询与数据获取方法

所用 `Query Builder` 支持的查询方法，例如 `select` 、 `selectSub` 、 `whereDate` 、 `whereBetween` 等等，都可以直接对 `Eloquent Model` 直接使用，程序会通过魔术方法调用 `Query Builder` 的相关方法：

```
protected $passthru = [
    'insert', 'insertGetId', 'getBindings', 'toSql',
    'exists', 'count', 'min', 'max', 'avg', 'sum', 'getConnectio
n',
];

public function __call($method, $parameters)
{
    ...

    if (in_array($method, $this->passthru)) {
        return $this->toBase()->{$method}(...$parameters);
    }

    $this->query->{$method}(...$parameters);

    return $this;
}
```

`passthru` 中的各个函数在调用前需要加载查询作用域，原因是这些操作基本上是 `aggregate` 的，需要添加搜索条件才能更加符合预期：

```
public function toBase()
{
    return $this->applyScopes()->getQuery();
}
```

添加和更新模型

save 函数

在 `Eloquent Model` 中，添加与更新模型可以统一用 `save` 函数。在添加模型的时候需要事先为 `model` 属性赋值，可以单个手动赋值，也可以批量赋值。在更新模型的时候，需要事先从数据库中取出模型，然后修改模型属性，最后执行 `save` 更新操作。官方文档：添加和更新模型

```
public function save(array $options = [])
{
    $query = $this->newQueryWithoutScopes();

    if ($this->fireModelEvent('saving') === false) {
        return false;
    }

    if ($this->exists) {
        $saved = $this->isDirty() ?
                    $this->performUpdate($query) : true;
    }

    else {
        $saved = $this->performInsert($query);

        if (! $this->getConnectionName() &&
            $connection = $query->getConnection()) {
            $this->setConnection($connection->getName());
        }
    }

    if ($saved) {
        $this->finishSave($options);
    }

    return $saved;
}
```

`save` 函数不会加载全局作用域，原因是凡是利用 `save` 函数进行的插入或者更新的操作都不会存在 `where` 条件，仅仅利用自身的主键属性来进行更新。如果需要 `where` 条件可以使用 `query\builder` 的 `update` 函数，我们在下面会详细介绍：

```
public function newQueryWithoutScopes()
{
    $builder = $this->newEloquentBuilder($this->newBaseQueryBuil
der());

    return $builder->setModel($this)
                ->with($this->with)
                ->withCount($this->withCount);
}

protected function newBaseQueryBuilder()
{
    $connection = $this->getConnection();

    return new QueryBuilder(
        $connection, $connection->getQueryGrammar(), $connection
->getPostProcessor()
    );
}
```

newQueryWithoutScopes 函数创建新的没有任何其他条件的 `Eloquent\builder` 类，而 `Eloquent\builder` 类需要 `Query\builder` 类作为底层查询构造器。

performUpdate 函数

如果当前的数据库模型对象是从数据库中取出的，也就是直接或间接的调用 `get()` 函数从数据库中获取到的数据库对象，那么其 `exists` 必然是 `true`

```
public function isDirty($attributes = null)
{
    return $this->hasChanges(
        $this->getDirty(), is_array($attributes) ? $attributes :
 func_get_args()
    );
}

public function getDirty()
{
    $dirty = [];

    foreach ($this->getAttributes() as $key => $value) {
        if (! $this->originalIsEquivalent($key, $value)) {
            $dirty[$key] = $value;
        }
    }

    return $dirty;
}
```

`getDirty` 函数可以获取所有与原始值不同的属性值，也就是需要更新的数据库字段。关键函数在于 `originalIsEquivalent` ：

```
protected function originalIsEquivalent($key, $current)
{
    if (! array_key_exists($key, $this->original)) {
        return false;
    }

    $original = $this->getOriginal($key);

    if ($current === $original) {
        return true;
    } elseif (is_null($current)) {
        return false;
    } elseif ($this->isDateAttribute($key)) {
        return $this->fromDateTime($current) ===
               $this->fromDateTime($original);
    } elseif ($this->hasCast($key)) {
        return $this->castAttribute($key, $current) ===
               $this->castAttribute($key, $original);
    }

    return is_numeric($current) && is_numeric($original)
            && strcmp((string) $current, (string) $original) ===
0;
}
```

可以看到，对于数据库可以转化的属性都要先进行转化，然后再开始对比。比较出的结果，就是我们需要 `update` 的字段。

执行更新的时候，除了 `getDirty` 函数获得的待更新字段，还会有 `UPDATED_AT` 这个字段：

```
protected function performUpdate(Builder $query)
{
    if ($this->fireModelEvent('updating') === false) {
        return false;
    }

    if ($this->usesTimestamps()) {
        $this->updateTimestamps();
    }

    $dirty = $this->getDirty();

    if (count($dirty) > 0) {
        $this->setKeysForSaveQuery($query)->update($dirty);

        $this->fireModelEvent('updated', false);

        $this->syncChanges();
    }

    return true;
}

protected function updateTimestamps()
{
    $time = $this->freshTimestamp();

    if (! is_null(static::UPDATED_AT) && ! $this->isDirty(static
::UPDATED_AT)) {
        $this->setUpdatedAt($time);
    }

    if (! $this->exists && ! $this->isDirty(static::CREATED_AT))
 {
        $this->setCreatedAt($time);
    }
}
```

执行更新的时候，`where` 条件只有一个，那就是主键 `id` ：

```
protected function setKeysForSaveQuery(Builder $query)
{
    $query->where($this->getKeyName(), '=', $this->getKeyForSave
Query());

    return $query;
}

protected function getKeyForSaveQuery()
{
    return $this->original[$this->getKeyName()]
                    ?? $this->getKey();
}

public function getKey()
{
    return $this->getAttribute($this->getKeyName());
}
```

最后会调用 `EloquentBuilder` 的 `update` 函数：

```
public function update(array $values)
{
    return $this->toBase()->update($this->addUpdatedAtColumn($va
lues));
}

protected function addUpdatedAtColumn(array $values)
{
    if (! $this->model->usesTimestamps()) {
        return $values;
    }

    return Arr::add(
        $values, $this->model->getUpdatedAtColumn(),
        $this->model->freshTimestampString()
    );
}

public function freshTimestampString()
{
    return $this->fromDateTime($this->freshTimestamp());
}

public function fromDateTime($value)
{
    return is_null($value) ? $value : $this->asDateTime($value)-
>format(
        $this->getDateFormat()
    );
}
```

performInsert

关于数据库对象的插入，如果数据库的主键被设置为 `increment` ，也就是自增的话，程序会调用 `insertAndSetId` ，这个时候不需要给数据库模型对象手动赋值主键 `id` 。若果数据库的主键并不支持自增，那么就需要在插入前，为数据库对象的主键 `id` 赋值，否则数据库会报错。

```
protected function performInsert(Builder $query)
{
    if ($this->fireModelEvent('creating') === false) {
        return false;
    }

    if ($this->usesTimestamps()) {
        $this->updateTimestamps();
    }

    $attributes = $this->attributes;

    if ($this->getIncrementing()) {
        $this->insertAndSetId($query, $attributes);
    }
    else {
        if (empty($attributes)) {
            return true;
        }

        $query->insert($attributes);
    }

    $this->exists = true;

    $this->wasRecentlyCreated = true;

    $this->fireModelEvent('created', false);

    return true;
}
```

`laravel` 默认数据库的主键支持自增属性，程序调用的也是函数 `insertAndSetId` 函数：

```
protected function insertAndSetId(Builder $query, $attributes)
{
    $id = $query->insertGetId($attributes, $keyName = $this->get
KeyName());

    $this->setAttribute($keyName, $id);
}
```

插入后，会将插入后得到的主键 `id` 返回，并赋值到模型的属性当中。

如果数据库主键不支持自增，那么我们在数据库类中要设置：

```
public $incrementing = false;
```

每次进行插入数据的时候，需要手动给主键赋值。

update 函数

`save` 函数仅仅支持手动的属性赋值，无法批量赋值。`laravel` 的 `Eloquent Model` 还有一个函数: `update` 支持批量属性赋值。有意思的是，`Eloquent Builder` 也有函数 `update`，那个是上一小节提到的 `performUpdate` 所调用的函数。

两个 `update` 功能一致，只是 `Model` 的 `update` 函数比较适用于更新从数据库取回的数据库对象：

```
$flight = App\Flight::find(1);

$flight->update(['name' => 'New Flight Name','desc' => 'test']);
```

而 `Builder` 的 `update` 适用于多查询条件下的更新：

```
App\Flight::where('active', 1)
          ->where('destination', 'San Diego')
          ->update(['delayed' => 1]);
```

无论哪一种，都会自动更新 `updated_at` 字段。

`Model` 的 `update` 函数借助 `fill` 函数与 `save` 函数：

```
public function update(array $attributes = [], array $options =
[])
{
    if (! $this->exists) {
        return false;
    }

    return $this->fill($attributes)->save($options);
}
```

make 函数

同样的，`save` 的插入也仅仅支持手动属性赋值，如果想实现批量属性赋值的插入可以使用 `make` 函数：

```
$model = App\Flight::make(['name' => 'New Flight Name','desc' =>
'test']);

$model->save();
```

`make` 函数实际上仅仅是新建了一个 `Eloquent Model`，并批量赋予属性值：

```
public function make(array $attributes = [])
{
    return $this->newModelInstance($attributes);
}

public function newModelInstance($attributes = [])
{
    return $this->model->newInstance($attributes)->setConnection
(
        $this->query->getConnection()->getName()
    );
}
```

create 函数

如果想要一步到位，批量赋值属性与插入一起操作，可以使用 `create` 函数：

```
App\Flight::create(['name' => 'New Flight Name','desc' => 'test'
]);
```

相比较 `make` 函数，`create` 函数更进一步调用了 `save` 函数：

```
public function create(array $attributes = [])
{
    return tap($this->newModelInstance($attributes), function ($
instance) {
        $instance->save();
    });
}
```

实际上，属性值是否可以批量赋值需要受 `fillable` 或 `guarded` 来控制，如果我们想要强制批量赋值可以使用 `forceCreate` ：

```
public function forceCreate(array $attributes)
{
    return $this->model->unguarded(function () use ($attributes)
{
        return $this->newModelInstance()->create($attributes);
    });
}
```

findOrNew 函数

`laravel` 提供一种主键查询或者新建数据库对象的函数：`findOrNew`：

```
public function findOrNew($id, $columns = ['*'])
{
    if (! is_null($model = $this->find($id, $columns))) {
        return $model;
    }

    return $this->newModelInstance();
}
```

值得注意的是，当查询失败的时候，会返回一个全新的数据库对象，不含有任何 `attributes`。

firstOrNew 函数

`laravel` 提供一种自定义查询或者新建数据库对象的函数：`firstOrNew`：

```
public function firstOrNew(array $attributes, array $values = [])

{
    if (! is_null($instance = $this->where($attributes)->first()
)) {
        return $instance;
    }

    return $this->newModelInstance($attributes + $values);
}
```

值得注意的是，如果查询失败，会返回一个含有 `attributes` 和 `values` 两者合并的属性的数据库对象。

firstOrCreate 函数

类似于 `firstOrNew` 函数，`firstOrCreate` 函数也用于自定义查询或者新建数据库对象，不同的是，`firstOrCreate` 函数还进一步对数据进行了插入操作：

```
public function firstOrCreate(array $attributes, array $values =
 [])
{
    if (! is_null($instance = $this->where($attributes)->first()
)) {
        return $instance;
    }

    return tap($this->newModelInstance($attributes + $values), f
unction ($instance) {
        $instance->save();
    });
}
```

updateOrCreate 函数

在 `firstOrCreate` 函数基础上，除了对数据进行查询，还会对查询成功的数据利用 `value` 进行更新：

```
public function updateOrCreate(array $attributes, array $values
= [])
{
    return tap($this->firstOrNew($attributes), function ($instan
ce) use ($values) {
        $instance->fill($values)->save();
    });
}
```

firstOr 函数

如果想要自定义查找失败后的操作，可以使用 `firstOr` 函数，该函数可以传入闭包函数，处理找不到数据的情况：

```
public function firstOr($columns = ['*'], Closure $callback = nu
ll)
{
    if ($columns instanceof Closure) {
        $callback = $columns;

        $columns = ['*'];
    }

    if (! is_null($model = $this->first($columns))) {
        return $model;
    }

    return call_user_func($callback);
}
```

删除模型

删除模型也会分为两种，一种是针对 `Eloquent Model` 的删除，这种删除必须是从数据库中取出的对象。还有一种是 `Eloquent Builder` 的删除，这种删除一般会带有多个查询条件。我们这一小节主要讲 `model` 的删除：

```
public function delete()
{
    if (is_null($this->getKeyName())) {
        throw new Exception('No primary key defined on model.');
    }

    if (! $this->exists) {
        return;
    }

    if ($this->fireModelEvent('deleting') === false) {
        return false;
    }

    $this->touchOwners();

    $this->performDeleteOnModel();

    $this->fireModelEvent('deleted', false);

    return true;
}
```

删除模型时，模型对象必然要有主键。`performDeleteOnModel` 函数执行具体的删除操作：

```
protected function performDeleteOnModel()
{
    $this->setKeysForSaveQuery($this->newQueryWithoutScopes())->
delete();

    $this->exists = false;
}

protected function setKeysForSaveQuery(Builder $query)
{
    $query->where($this->getKeyName(), '=', $this->getKeyForSave
Query());

    return $query;
}
```

所以实际上，`Model` 调用的也是 `builder` 的 `delete` 函数。

软删除

如果想要使用软删除，需要使用 `Illuminate\Database\Eloquent\SoftDeletes` 这个 trait。并且需要定义软删除字段，默认为 `deleted_at`，将软删除字段放入 `dates` 中，具体用法可参考官方文档:软删除

```
class Flight extends Model
{
    use SoftDeletes;

    /**
     * 需要被转换成日期的属性。
     *
     * @var array
     */
    protected $dates = ['deleted_at'];
}
```

我们先看看这个 `trait` ：

```
trait SoftDeletes
{
    public static function bootSoftDeletes()
    {
        static::addGlobalScope(new SoftDeletingScope);
    }

}
```

如果使用了软删除，在 `model` 的启动过程中，就会启动软删除的这个函数。可以看出来，软删除是需要查询作用域来合作发挥作用的。我们看看这个 `SoftDeletingScope` ：

```
class SoftDeletingScope implements Scope
{
    protected $extensions = ['Restore', 'WithTrashed', 'WithoutT
rashed', 'OnlyTrashed'];

    public function apply(Builder $builder, Model $model)
    {
        $builder->whereNull($model->getQualifiedDeletedAtColumn(
));
    }

    public function extend(Builder $builder)
    {
        foreach ($this->extensions as $extension) {
            $this->{"add{$extension}"}($builder);
        }

        $builder->onDelete(function (Builder $builder) {
            $column = $this->getDeletedAtColumn($builder);

            return $builder->update([
                $column => $builder->getModel()->freshTimestampS
tring(),
            ]);
        });
    }
}
```

`apply` 函数是加载全局域调用的函数，每次进行查询的时候，调用 `get` 函数就会自动加载这个函数，`whereNull` 这个查询条件会被加载到具体的 `where` 条件中。`deleted_at` 字段一般被设置为 `null`，在执行软删除的时候，该字段会被赋予时间格式的值，标志着被删除的时间。

在加载全局作用域的时候，还会调用 `extend` 函数，`extend` 函数为 `model` 添加了四个函数：

- WithTrashed

```
protected function addWithTrashed(Builder $builder)
{
    $builder->macro('withTrashed', function (Builder $builder) {
        return $builder->withoutGlobalScope($this);
    });
}
```

`withTrashed` 函数取消了软删除的全局作用域，这样我们查询数据的时候就会查询到正常数据和被软删除的数据。

- withoutTrashed

```
protected function addWithoutTrashed(Builder $builder)
{
    $builder->macro('withoutTrashed', function (Builder $builder)
 {
        $model = $builder->getModel();

        $builder->withoutGlobalScope($this)->whereNull(
            $model->getQualifiedDeletedAtColumn()
        );

        return $builder;
    });
}
```

`withTrashed` 函数着重强调了不要获取软删除的数据。

- onlyTrashed

```
protected function addOnlyTrashed(Builder $builder)
{
    $builder->macro('onlyTrashed', function (Builder $builder) {
        $model = $builder->getModel();

        $builder->withoutGlobalScope($this)->whereNotNull(
            $model->getQualifiedDeletedAtColumn()
        );

        return $builder;
    });
}
```

如果只想获取被软删除的数据，可以使用这个函数 `onlyTrashed` ，可以看到，它使用了 `whereNotNull` 。

- restore

```
protected function addRestore(Builder $builder)
{
    $builder->macro('restore', function (Builder $builder) {
        $builder->withTrashed();

        return $builder->update([$builder->getModel()->getDelete
dAtColumn() => null]);
    });
}
```

如果想要恢复被删除的数据，还可以使用 `restore` ，重新将 `deleted_at` 数据恢复为 null。

performDeleteOnModel

SoftDeletes 这个 trait 会重载 `performDeleteOnModel` 函数，它将不会调用 `Eloquent Builder` 的 `delete` 方法，而是采用更新操作：

```
protected function performDeleteOnModel()
{
    if ($this->forceDeleting) {
        return $this->newQueryWithoutScopes()->where($this->getKeyName(), $this->getKey())->forceDelete();
    }

    return $this->runSoftDelete();
}

protected function runSoftDelete()
{
    $query = $this->newQueryWithoutScopes()->where($this->getKeyName(), $this->getKey());

    $time = $this->freshTimestamp();

    $columns = [$this->getDeletedAtColumn() => $this->fromDateTime($time)];

    $this->{$this->getDeletedAtColumn()} = $time;

    if ($this->timestamps) {
        $this->{$this->getUpdatedAtColumn()} = $time;

        $columns[$this->getUpdatedAtColumn()] = $this->fromDateTime($time);
    }

    $query->update($columns);
}
```

删除操作不仅更新了 `deleted_at`，还更新了 `updated_at` 字段。

Laravel Database——Eloquent Model 关联源码分析

前言

数据库表通常相互关联。 `laravel` 中的模型关联功能使得关于数据库的关联代码变得更加简单，更加优雅。本文会详细说说关于模型关联的源码，以便更好的理解和使用关联模型。官方文档：Eloquent：关联

定义关联

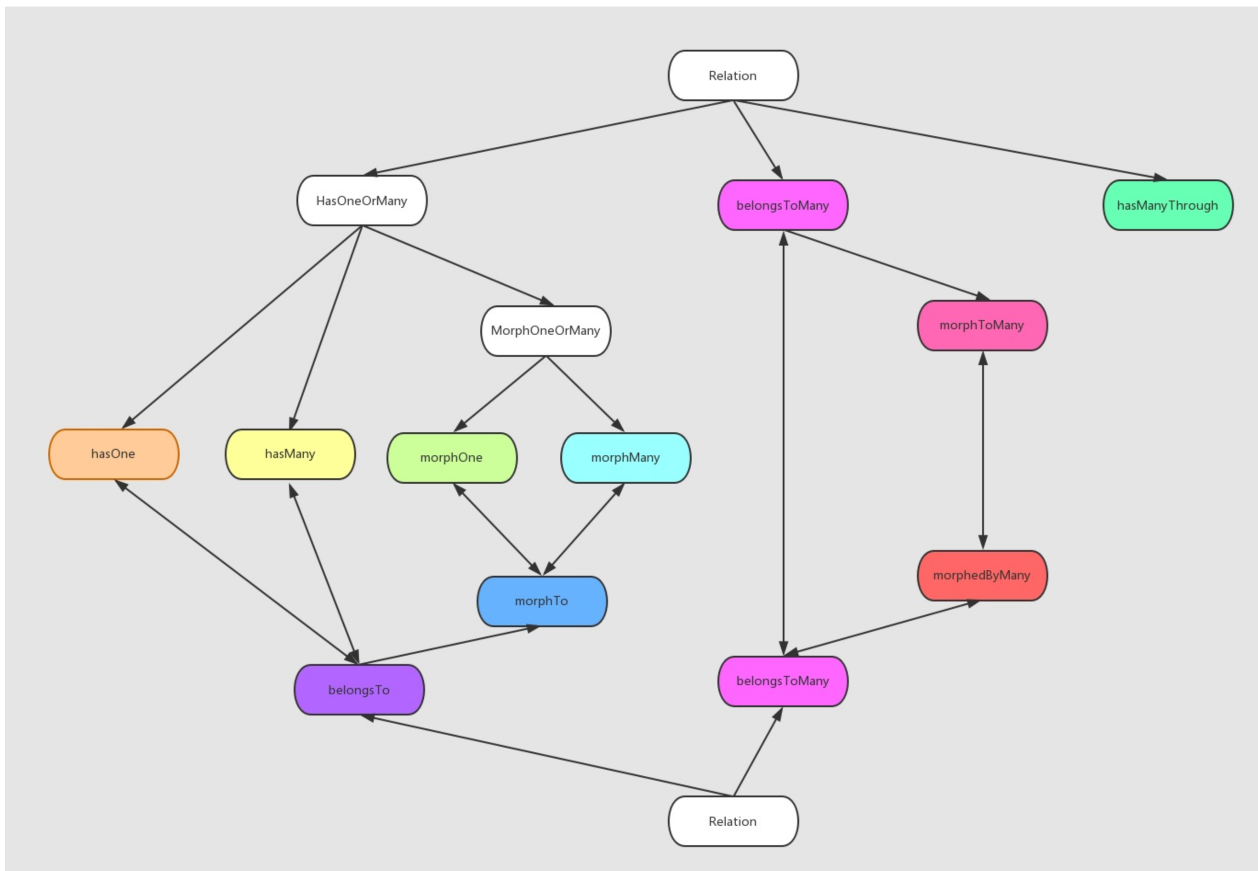
所谓的定义关联，就是在一个 `Model` 中定义一个关联函数，我们利用这个关联函数去操作另外一个 `Model` ，例如， `user` 表是用户表， `posts` 是用户发的文章，一个用户可以发表多篇文章，我们就可以这样写：

```
$user->posts()->where('active', 1)->get();
```

这表明了我们想通过 `$user` 这个用户查询到状态 `active` 为 1 的所有文章， `posts` 就是关联函数，我们可以通过这个关联函数去操作另一个与 `user` 关联的表。

在说模型关联的定义之前，我们要先说说父模型与子模型的概念。所谓的父模型是指在模型关系中主动的一方，例如用户模型和文章模型中的用户，相应的子模型就是模型关系中的被动一方，例如文章模型。在正向定义中，被关联的是子模型，而在反向关联中，被关联的是父模型。

我们知道，关联有多种形式，各种关系如下：



hasOne 一对一

我们以官方文档的例子来说明，一个 `User` 模型可能关联一个 `Phone` 模型：

```
class User extends Model
{
    /**
     * 获得与用户关联的电话记录。
     */
    public function phone()
    {
        $this->hasOne('App\Phone', 'user_id', 'id');
    }
}
```

我们来看看 `hasOne` 的源码：

```
public function hasOne($related, $foreignKey = null, $localKey = null)
{
    $instance = $this->newRelatedInstance($related);

    $foreignKey = $foreignKey ?: $this->getForeignKey();

    $localKey = $localKey ?: $this->getKeyName();

    return new HasOne($instance->newQuery(), $this, $instance->getTable().'.'.$foreignKey, $localKey);
}
```

`newRelatedInstance` 函数负责建立一个新的被关联的模型实例，主要目的是设置数据库连接：

```
protected function newRelatedInstance($class)
{
    return tap(new $class, function ($instance) {
        if (! $instance->getConnectionName()) {
            $instance->setConnection($this->connection);
        }
    });
}
```

在一对一的关系中，`foreignKey` 外键名默认是父模型的类名和主键名的蛇形变量，`localKey` 是父模型的主键名：

```
public function getForeignKey()
{
    return Str::snake(class_basename($this)).'_'.$this->primaryKey;
}
```

`hasOne` 函数的构造函数继承 `HasOneOrMany` 类，也就是说，一对一与一对多构造函数相同，这部分主要设置外键名：

```
public function __construct(Builder $query, Model $parent, $fore
ignKey, $localKey)
{
    $this->localKey = $localKey;
    $this->foreignKey = $foreignKey;

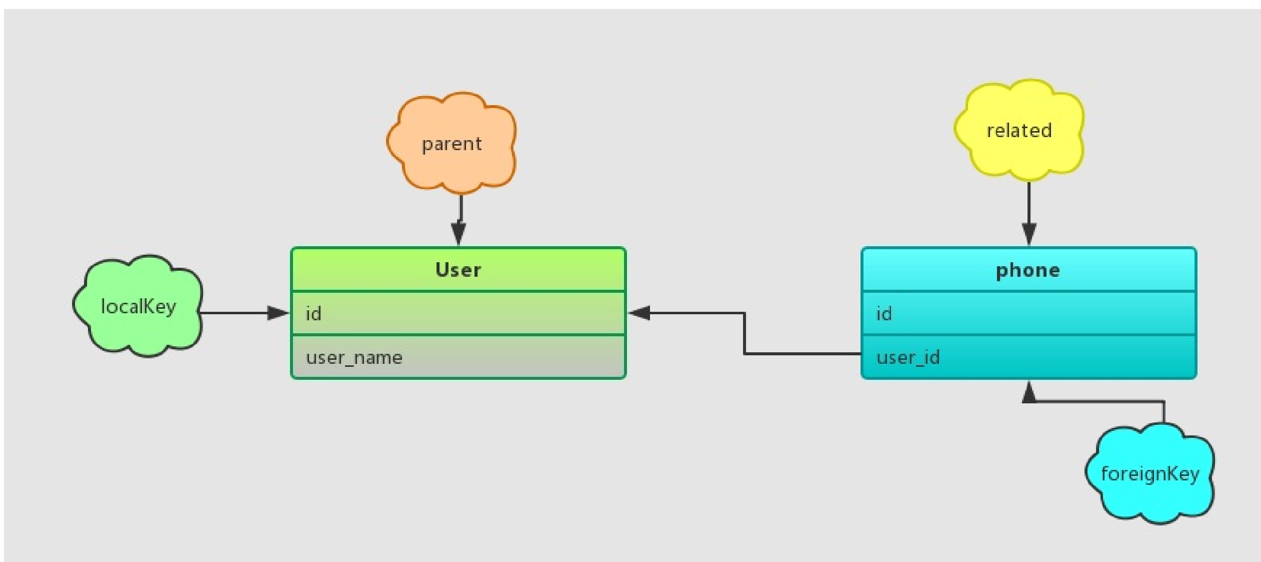
    parent::__construct($query, $parent);
}
```

`HasOneOrMany` 类继承 `Relation` 类，这部分主要设置 `parent` (父模型)、被关联模型（子模型）与被关联模型（子模型）的查询构造器：

```
public function __construct(Builder $query, Model $parent)
{
    $this->query = $query;
    $this->parent = $parent;
    $this->related = $query->getModel();

    $this->addConstraints();
}
```

`hasOne` 的模型关系如下：



除了保存被关联模型的查询构造器、被关联模型与 `parent` 模型之外，还会提供额外的限制条件：

```
public function addConstraints()
{
    if (static::$constraints) {
        $this->query->where($this->foreignKey, '=', $this->getPa
rentKey());

        $this->query->whereNotNull($this->foreignKey);
    }
}

public function getParentKey()
{
    return $this->parent->getAttribute($this->localKey);
}
```

限制条件为被关联模型和关联模型建立外键约束关系：

```
select phone where phone.user_id = 1 (user.id)
```

hasMany 一对多

在模型关联的定义中，一对一与一对多源码是一样的：

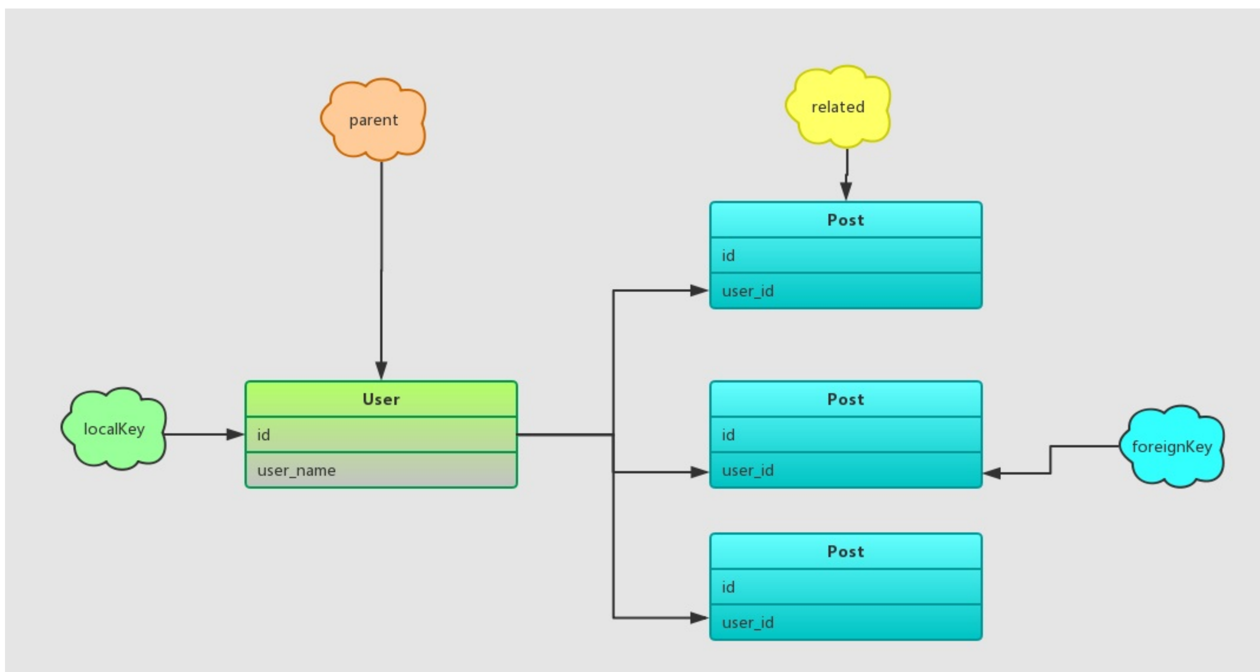
```
public function hasMany($related, $foreignKey = null, $localKey
= null)
{
    $instance = $this->newRelatedInstance($related);

    $foreignKey = $foreignKey ?: $this->getForeignKey();

    $localKey = $localKey ?: $this->getKeyName();

    return new HasMany(
        $instance->newQuery(), $this, $instance->getTable().'.'.
$foreignKey, $localKey
    );
}
```

`hasMany` 的模型关系如下：



限制条件与一对一相同，为被关联模型和关联模型建立外键约束关系：

```
select phone where phone.user_id = 1 (user.id)
```

belongsTo 一对一、一对多 反向关联

如果想要从文章反向查找作者用户，那么可以定义反向关联：

```
public function user()
{
    return $this->belongsTo('App\User', 'foreign_key', 'other_ke
y');
}
```

`belongsTo` 源码：

```
public function belongsTo($related, $foreignKey = null, $ownerKe
y = null, $relation = null)
{
    if (is_null($relation)) {
        $relation = $this->guessBelongsToRelation();
    }

    $instance = $this->newRelatedInstance($related);

    if (is_null($foreignKey)) {
        $foreignKey = Str::snake($relation).'_'.$instance->getKe
yName();
    }

    $ownerKey = $ownerKey ?: $instance->getKeyName();

    return new BelongsTo(
        $instance->newQuery(), $this, $foreignKey, $ownerKey, $r
elation
    );
}
```

正向定义与反向定义不同的是多了一个参数 `relation` ，这个参数默认值是从 `debug_backtrace` 函数获取的：

```
protected function guessBelongsToRelation()
{
    list($one, $two, $caller) = debug_backtrace(DEBUG_BACKTRACE_
IGNORE_ARGS, 3);

    return $caller['function'];
}
```

也就是我们的关联函数名 `user`，`belongsTo` 函数会将关联函数名作为关联名保存起来。

另一个不同是外键的默认名称，不再是类名 + 主键名，而是关联名 + 主键名：

```
if (is_null($foreignKey)) {
    $foreignKey = Str::snake($relation).'_'.$instance->getKeyNam
e();
}
```

我们接着看 `belongsTo` 函数：

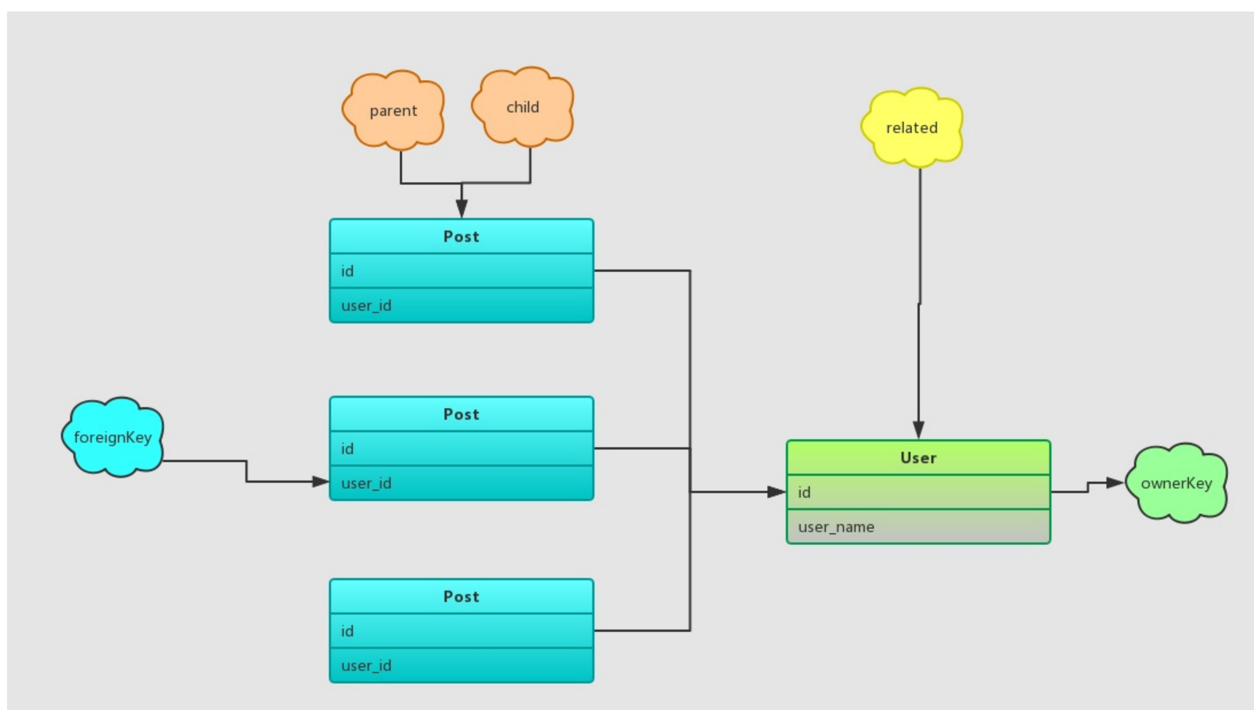
```
public function __construct(Builder $query, Model $child, $forei
gnKey, $ownerKey, $relation)
{
    $this->ownerKey = $ownerKey;
    $this->relation = $relation;
    $this->foreignKey = $foreignKey;

    $this->child = $child;

    parent::__construct($query, $child);
}
```

我们可以看出来，相对于正向关联，反向关联除了保存外键名与主键名之外，还保存了关系名、子模型。值得注意的是，反向关联中 `related` 代表父模型，`parent` 代表子模型，与正向关联相反。

`hasMany` 的模型关系如下：



约束条件也相应地进行反转改变：

```
public function addConstraints()
{
    if (static::$constraints) {
        $table = $this->related->getTable();

        $this->query->where($table.'.'.$this->ownerKey, '=', $th
is->child->{$this->foreignKey});
    }
}
```

限制条件：

```
select user where user.id = 1 (post.user_id)
```

belongsMany 多对多

多对多关系由于中间表的原因相对来说比较复杂，涉及的参数也非常多。我们以官网例子：

```
class User extends Model
{
    /**
     * 获得此用户的角色。
     */
    public function roles()
    {
        return $this->belongsToMany('App\Role', 'role_user', 'us
er_id', 'role_id');
    }
}
```

`User` 表与 `role` 表是多对多关系，另外有一中间表 `user_role` 表，我们在定义关系的时候，`related` 是被关联模型，`table` 是中间表，`foreignPivotKey` 是中间表中父模型外键名，`relatedPivotKey` 是中间表中子模型外键名，`parentKey` 是父模型主键名，`relatedKey` 是子模型主键名，`relation` 是关系名。

```
public function belongsToMany($related, $table = null, $foreignP
ivotKey = null, $relatedPivotKey = null, $parentKey = null, $rel
atedKey = null, $relation = null)
{
    if (is_null($relation)) {
        $relation = $this->guessBelongsToManyRelation();
    }

    $instance = $this->newRelatedInstance($related);

    $foreignPivotKey = $foreignPivotKey ?: $this->getForeignKey(
);

    $relatedPivotKey = $relatedPivotKey ?: $instance->getForeign
Key();

    if (is_null($table)) {
        $table = $this->joiningTable($related);
    }

    return new BelongsToMany(
        $instance->newQuery(), $this, $table, $foreignPivotKey,
        $relatedPivotKey, $parentKey ?: $this->getKeyName(),
        $relatedKey ?: $instance->getKeyName(), $relation
    );
}
```

获取关联名称仍然使用的是 `debug_backtrace` 函数，不同于 `guessBelongsToRelation` 函数只有 `belongsTo` 调用，`guessBelongsToManyRelation` 函数还可以被 `morphedByMany` 函数调用，所以不能单纯的限制返回堆栈帧：

```
public static $manyMethods = [
    'belongsToMany', 'morphToMany', 'morphedByMany',
    'guessBelongsToManyRelation', 'findFirstMethodThatIsntRelati
on',
];

protected function guessBelongsToManyRelation()
{
    $caller = Arr::first(debug_backtrace(DEBUG_BACKTRACE_IGNORE_
ARGS), function ($trace) {
        return ! in_array($trace['function'], Model::$manyMethod
s);
    });

    return ! is_null($caller) ? $caller['function'] : null;
}
```

默认的中间表是两个表名的蛇形变量：

```
public function joiningTable($related)
{
    $models = [
        Str::snake(class_basename($related)),
        Str::snake(class_basename($this)),
    ];

    sort($models);

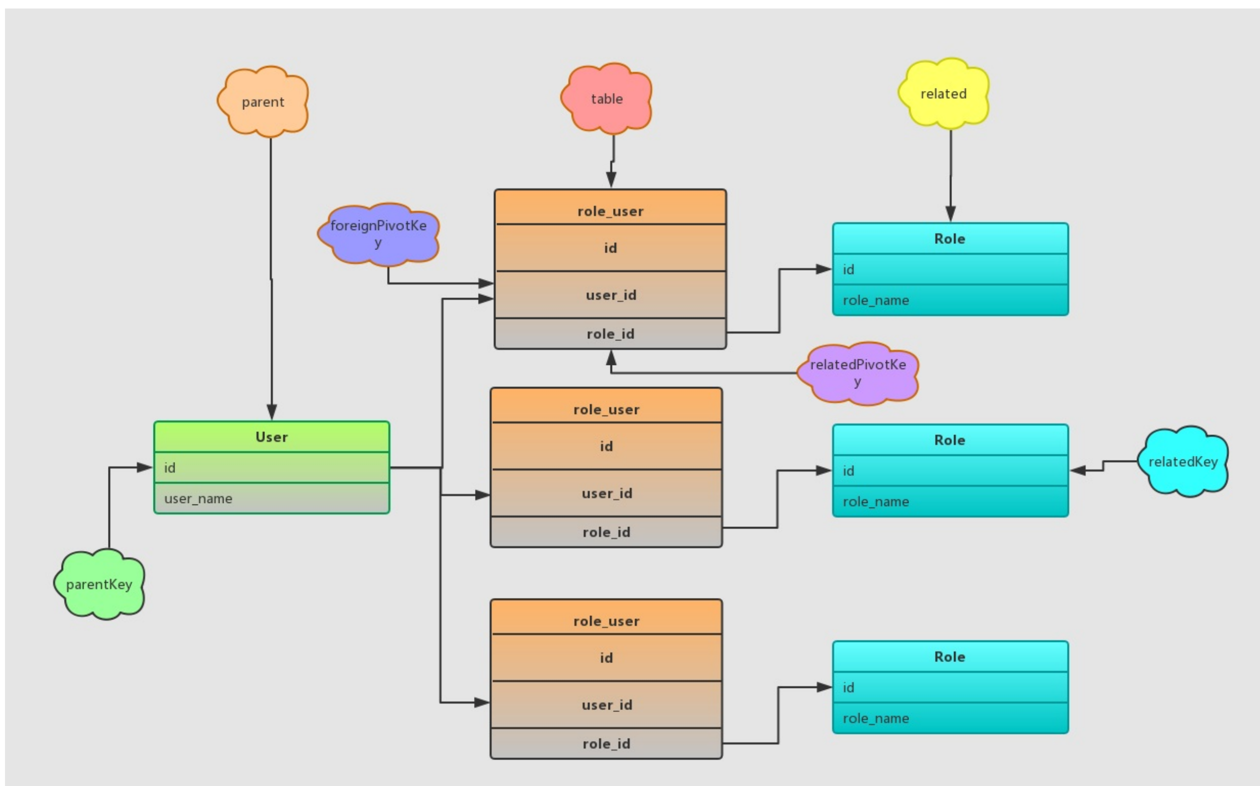
    return strtolower(implode('_', $models));
}
```

`BelongsToMany` 的初始化也需要保存这些变量：

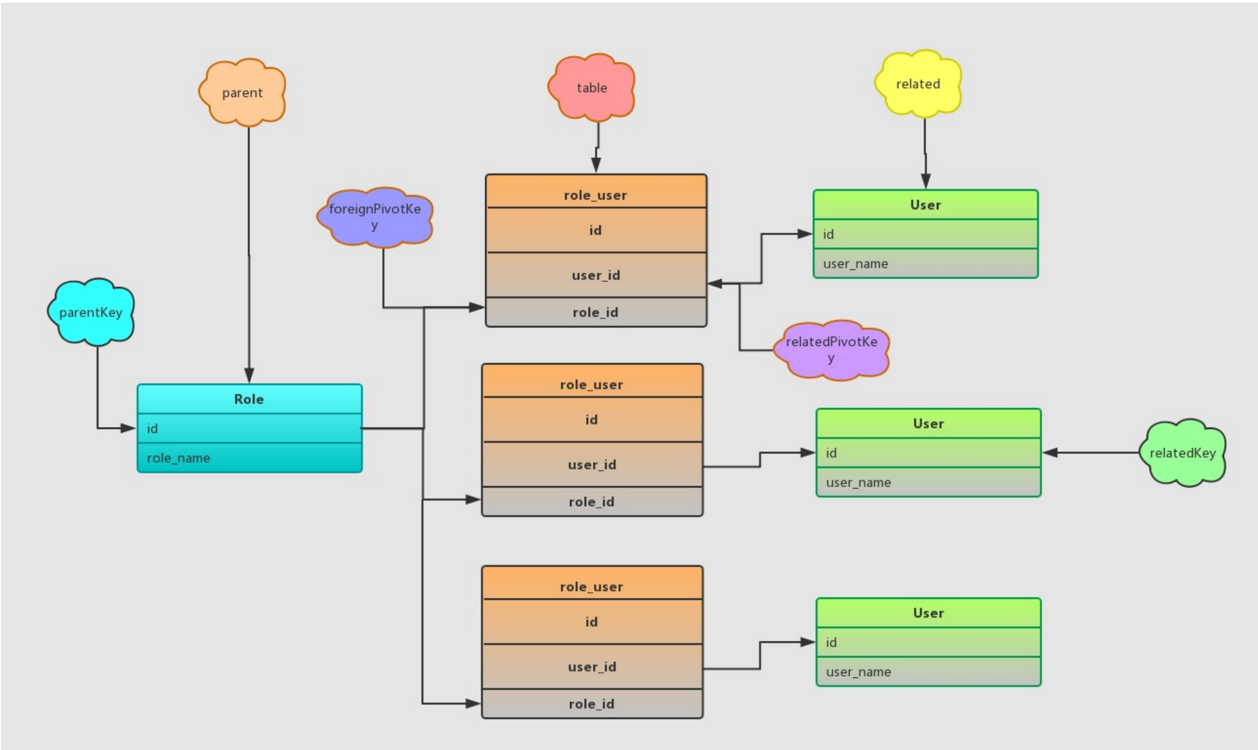
```
public function __construct(Builder $query, Model $parent, $tabl
e, $foreignPivotKey,
                                $relatedPivotKey, $parentKey, $r
elatedKey, $relationName = null)
{
    $this->table = $table;
    $this->parentKey = $parentKey;
    $this->relatedKey = $relatedKey;
    $this->relationName = $relationName;
    $this->relatedPivotKey = $relatedPivotKey;
    $this->foreignPivotKey = $foreignPivotKey;

    parent::__construct($query, $parent);
}
```

`belongsToMany` 的模型关系如下：



反向的多对多模型关系：


parent
table
related
foreignPivotKe
y
role_user
id
user_id
role_id
User
id
user_name
parentKey
relatedPivotKe
y
Role
id
role_name
role_user
id
user_id
role_id
User
id
user_name
relatedKey
role_user
id
user_id
role_id
User
id
user_name

限制条件：

```
public function addConstraints()
{
    $this->performJoin();

    if (static::$constraints) {
        $this->addWhereConstraints();
    }
}

protected function performJoin($query = null)
{
    $query = $query ?: $this->query;

    $baseTable = $this->related->getTable();

    $key = $baseTable.'.'.$this->relatedKey;

    $query->join($this->table, $key, '=', $this->getQualifiedRel
atedPivotKeyName());

    return $this;
}

protected function addWhereConstraints()
{
    $this->query->where(
        $this->getQualifiedForeignPivotKeyName(), '=', $this->pa
rent->{$this->parentKey}
    );

    return $this;
}
```

本例中的 where 条件：

```
select role join role_user on role_user.role_id = 1 (role.id)

select role where role_user.user_id = 1 (user.id)
```

hasManyThrough 远程一对多

远层一对多 关联提供了方便、简短的方式通过中间的关联来获得远层的关联。以官方例子来看：

```
class Country extends Model
{
    public function posts()
    {
        return $this->hasManyThrough(
            'App\Post',
            'App\User',
            'country_id', // 用户表外键...
            'user_id', // 文章表外键...
            'id', // 国家表本地键...
            'id' // 用户表本地键...
        );
    }
}
```

可以看到，远程一对多的参数比较多。第一个参数 `related` 是最终被关联的模型，`through` 是中间模型，`firstKey` 是中间模型关于父模型的外键，`secondKey` 是最终被关联的模型关于中间模型的外键，`localKey` 是父模型的主键，`secondLocalKey` 是中间模型的主键：

```
public function hasManyThrough($related, $through, $firstKey = n
ull, $secondKey = null, $localKey = null, $secondLocalKey = null)

{
    $through = new $through;

    $firstKey = $firstKey ?: $this->getForeignKey();

    $secondKey = $secondKey ?: $through->getForeignKey();

    $localKey = $localKey ?: $this->getKeyName();

    $secondLocalKey = $secondLocalKey ?: $through->getKeyName();

    $instance = $this->newRelatedInstance($related);

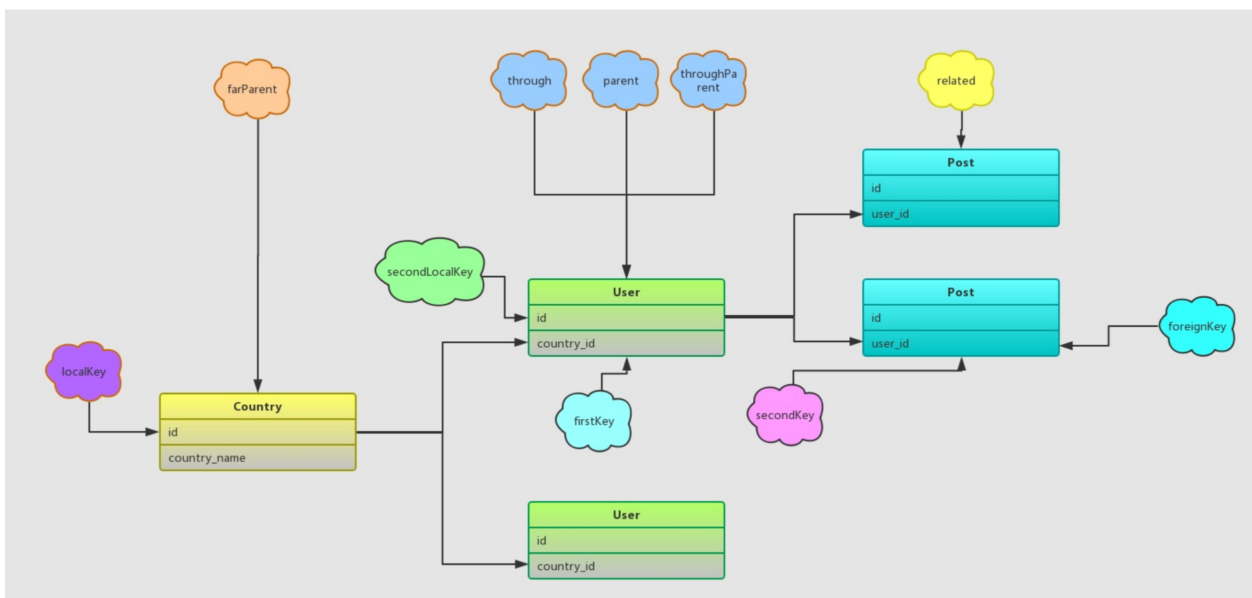
    return new HasManyThrough($instance->newQuery(), $this, $thr
ough, $firstKey, $secondKey, $localKey, $secondLocalKey);
}
```

HasManyThrough 的初始化：

```
public function __construct(Builder $query, Model $farParent, Mo
del $throughParent, $firstKey, $secondKey, $localKey, $secondLoc
alKey)
{
    $this->localKey = $localKey;
    $this->firstKey = $firstKey;
    $this->secondKey = $secondKey;
    $this->farParent = $farParent;
    $this->throughParent = $throughParent;
    $this->secondLocalKey = $secondLocalKey;

    parent::__construct($query, $throughParent);
}
```

`hasManyThrough` 的模型关系如下：



限制条件：

```
public function addConstraints()
{
    $localValue = $this->farParent[$this->localKey];

    $this->performJoin();

    if (static::$constraints) {
        $this->query->where($this->getQualifiedFirstKeyName(), '
=', $localValue);
    }
}

protected function performJoin(Builder $query = null)
{
    $query = $query ?: $this->query;

    $farKey = $this->getQualifiedFarKeyName();

    $query->join($this->throughParent->getTable(), $this->getQua
lifiedParentKeyName(), '=', $farKey);

    if ($this->throughParentSoftDeletes()) {
        $query->whereNull($this->throughParent->getQualifiedDele
tedAtColumn());
    }
```

```
}

public function getQualifiedParentKeyName()
{
    return $this->parent->getTable().'.'.$this->secondLocalKey;
}

public function getQualifiedFarKeyName()
{
    return $this->getQualifiedForeignKeyName();
}

public function getQualifiedForeignKeyName()
{
    return $this->related->getTable().'.'.$this->secondKey;
}

public function getQualifiedFirstKeyName()
{
    return $this->throughParent->getTable().'.'.$this->firstKey;
}
```

本例中的限制条件：

```

select post join user on user.id = post.user_id

select post where user.delete_at is null

select post where user.country_id = 1 (country.id)
```

morphOne/morphMany 多态关联

多态关联允许我们应用一个表来单独作为多个表的属性，多态关联存在一对一、一对多、多对多的情形。所谓一对一、一对多是指，一个模型只拥有一个属性或多个属性，例如官网中的例子：

> 用户可以「评论」文章和视频。使用多态关联，您可以用一个 comments 表同时满足这两个使用场景

```
class Post extends Model
{
    /**
     * 获得此文章的所有评论。
     */
    public function comments()
    {
        return $this->morphMany('App\Comment', 'commentable');
    }
}

class Video extends Model
{
    /**
     * 获得此视频的所有评论。
     */
    public function comments()
    {
        return $this->morphMany('App\Comment', 'commentable');
    }
}
```

这个 `comments` 表就是属性表，当文章和视频只能有一个评论的时候，那么就是一对一多态关联；如果文章和视频可以由多个评论的时候，就是一对多多态关联。

这种属性表一般会有两个固定的字段：`commentable_type` 用于标识该条评论是文章的还是视频的、`commentable_id` 用于记录文章或视频的主键 `id`。

我们可以把多态关联看作普通的一对一、一对多关系，只是外键参数是 `type` 与 `id` 的组合。

`related` 是属性表，也就是这里的 `comments`，`type` 参数是属性表中存储父模型类型的列名(commentable_type)，`id` 参数是属性表中存储父模型主键的列名(commentable_id)，而 `name` 专用于省略 `type` 参数与 `id` 参数，`localKey` 是指父模型的主键。

```
public function morphOne($related, $name, $type = null, $id = nu
ll, $localKey = null)
{
    $instance = $this->newRelatedInstance($related);

    list($type, $id) = $this->getMorphs($name, $type, $id);

    $table = $instance->getTable();

    $localKey = $localKey ?: $this->getKeyName();

    return new MorphOne($instance->newQuery(), $this, $table.'.'
.$type, $table.'.'.$id, $localKey);
}

public function morphMany($related, $name, $type = null, $id = n
ull, $localKey = null)
{
    $instance = $this->newRelatedInstance($related);

    list($type, $id) = $this->getMorphs($name, $type, $id);

    $table = $instance->getTable();

    $localKey = $localKey ?: $this->getKeyName();

    return new MorphMany($instance->newQuery(), $this, $table.'.'
.$type, $table.'.'.$id, $localKey);
}

protected function getMorphs($name, $type, $id)
{
    return [$type ?: $name.'_type', $id ?: $name.'_id'];
}
```

一对一、一对多多态关联主要保存属性表中表示类型的列名，还有需要向该类型列中写入的父模型名称，一般来说，默认会写父模型的类名（`App\Post`、`App\Video`）

```
public function __construct(Builder $query, Model $parent, $type
, $id, $localKey)
{
    $this->morphType = $type;

    $this->morphClass = $parent->getMorphClass();

    parent::__construct($query, $parent, $id, $localKey);
}

public function getMorphClass()
{
    $morphMap = Relation::morphMap();

    if (! empty($morphMap) && in_array(static::class, $morphMap)
) {
        return array_search(static::class, $morphMap, true);
    }

    return static::class;
}
```

不过我们也可以自定义写入的值：

```
Relation::morphMap([
    'posts' => 'App\Post',
    'videos' => 'App\Video',
]);
```

这样，就会把 `App\Post` 换成 `posts` ，`App\Video` 换成 `videos` 。我们来看看这个 `多态映射表` 函数：

```
public static function morphMap(array $map = null, $merge = true)

{
    $map = static::buildMorphMapFromModels($map);

    if (is_array($map)) {
        static::$morphMap = $merge && static::$morphMap
                        ? array_merge(static::$morphMap, $map) :
 $map;
    }

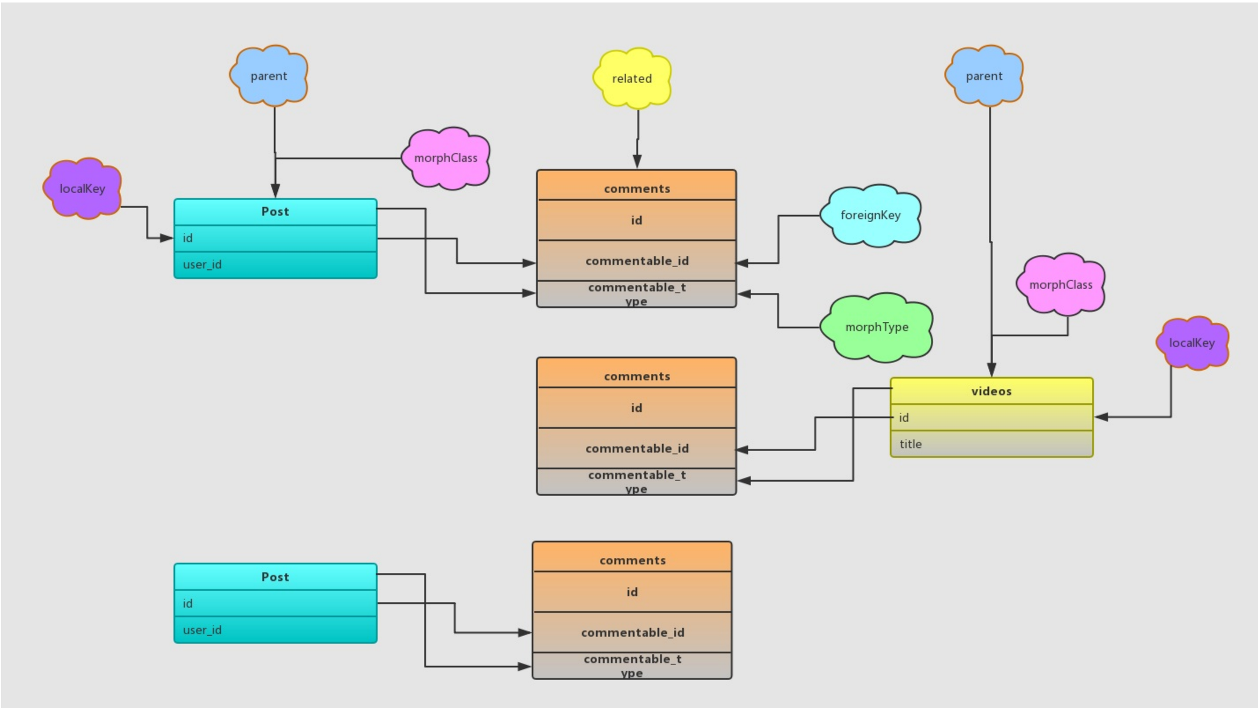
    return static::$morphMap;
}

protected static function buildMorphMapFromModels(array $models
= null)
{
    if (is_null($models) || Arr::isAssoc($models)) {
        return $models;
    }

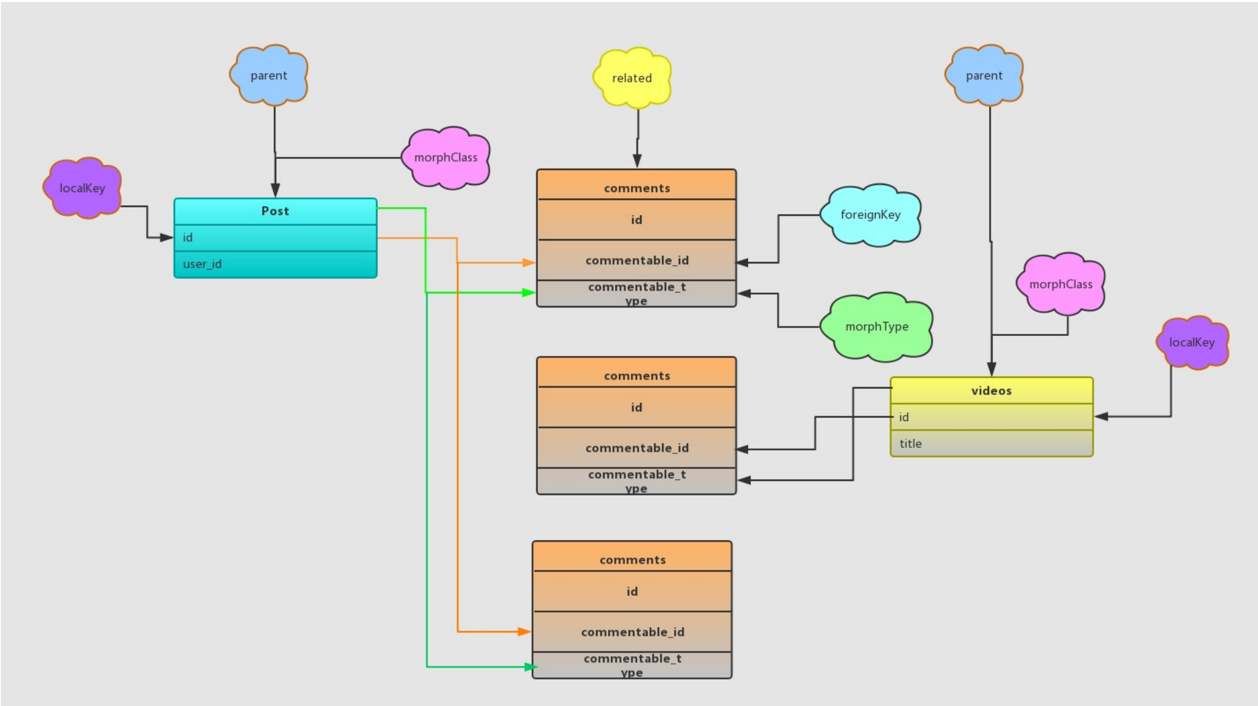
    return array_combine(array_map(function ($model) {
        return (new $model)->getTable();
    }, $models), $models);
}
```

可以看到，`buildMorphMapFromModels` 函数将字符串 `App\Post` 转为 `model`，并利用 `array_combine` 转为键。

`morphOne` 的模型关系如下：



`morphMany` 的模型关系如下：



限制条件：

```
public function addConstraints()
{
    if (static::$constraints) {
        parent::addConstraints();

        $this->query->where($this->morphType, $this->morphClass)
;
    }
}

public function addConstraints()
{
    if (static::$constraints) {
        $this->query->where($this->foreignKey, '=', $this->getPa
rentKey());

        $this->query->whereNotNull($this->foreignKey);
    }
}
```

本例中的限制条件：

```

select comments where comment.commentable_id = post.id

select comments where comment.commentable_id is not null

select comments where comment.commentable_type = 'App\Post'
```

morphTo 反向多态关联

和一对一、一对多的 `belongsTo` 相似，多态关联还可以定义反向关联 `morphTo`：

```
class Comment extends Model
{
    /**
     * 获得拥有此评论的模型。
     */
    public function commentable()
    {
        return $this->morphTo();
    }
}
```

与 `belongsTo` 类似， `morphTo` 也是利用 `debug_backtrace` 获取关联名称。当前如果正处于预加载状态的时候， `Comment` 一般还没有从数据库获取数据， `$this->{$type}` 是空值，这个时候需要去除预加载来初始化：

```
public function morphTo($name = null, $type = null, $id = null)
{
    $name = $name ?: $this->guessBelongsToRelation();

    list($type, $id) = $this->getMorphs(
        Str::snake($name), $type, $id
    );

    return empty($class = $this->{$type})
                ? $this->morphEagerTo($name, $type, $id)
                : $this->morphInstanceTo($class, $name, $type, $
id);
}

protected function morphEagerTo($name, $type, $id)
{
    return new MorphTo(
        $this->newQuery()->setEagerLoads([]), $this, $id, null,
$type, $name
    );
}

protected function morphInstanceTo($target, $name, $type, $id)
{
    $instance = $this->newRelatedInstance(
        static::getActualClassNameForMorph($target)
    );

    return new MorphTo(
        $instance->newQuery(), $this, $id, $instance->getKeyName
(), $type, $name
    );
}
```

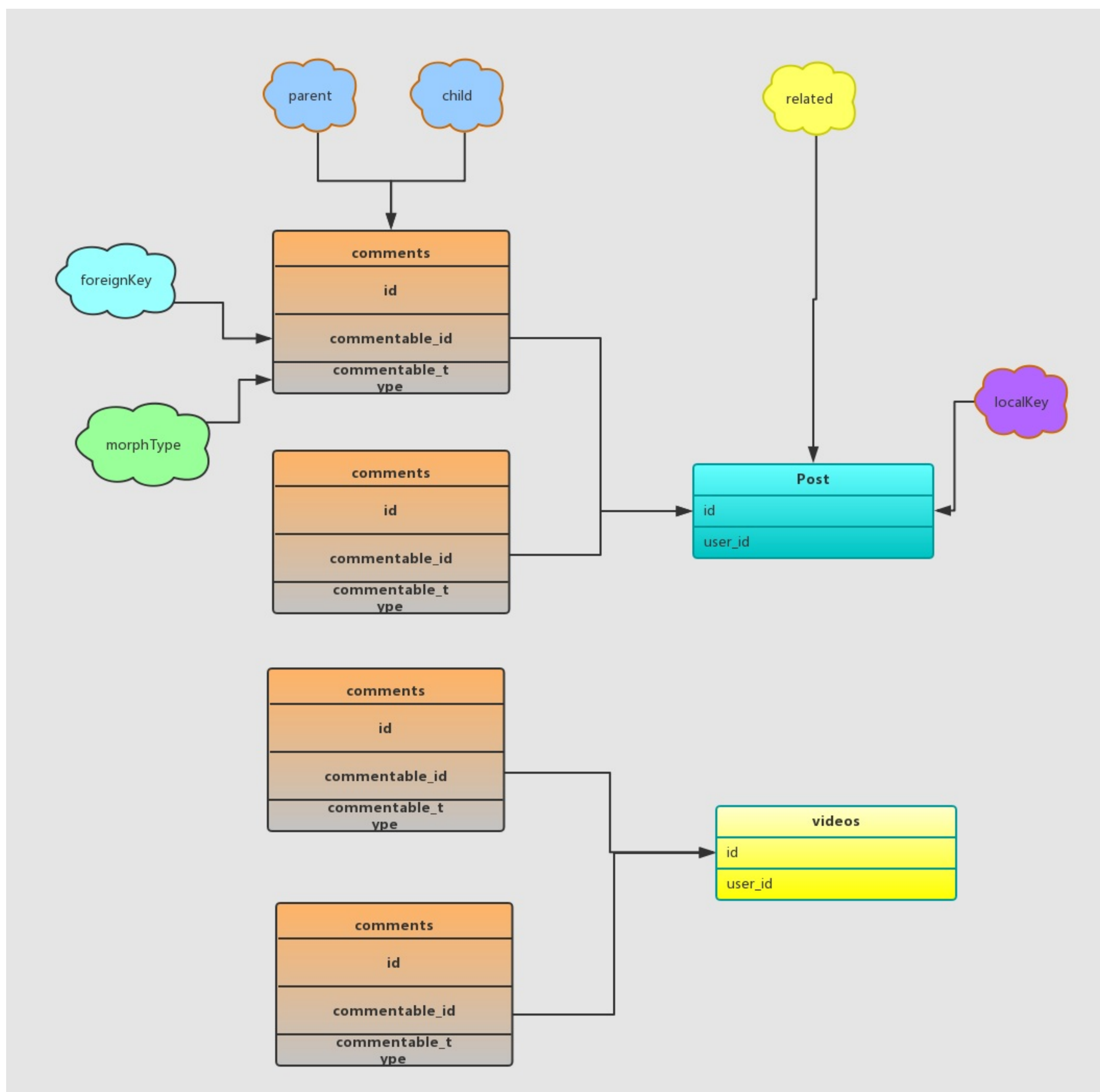
多态的成员变量 `morphType` 代表属性表的类型列，`morphClass`

`MorphTo` 的成员变量只有一个 `morphType`：

```
public function __construct(Builder $query, Model $parent, $fore
ignKey, $ownerKey, $type, $relation)
{
    $this->morphType = $type;

    parent::__construct($query, $parent, $foreignKey, $ownerKey,
 $relation);
}
```

`morphTo` 的模型关系如下：



限制条件与 `belongsTo` 相同：

```
public function addConstraints()
{
    if (static::$constraints) {
        $table = $this->related->getTable();

        $this->query->where($table.'.'.$this->ownerKey, '=', $th
is->child->{$this->foreignKey});
    }
}
```

本例中的限制条件

```

select post where post.id = comments.commentable_id
```

多对多多态关联

除了传统的多态关联，您也可以定义「多对多」的多态关联。例如，Post 模型和 Video 模型可以共享一个多态关联至 Tag 模型。使用多对多多态关联可以让您在文章和视频中共享唯一的标签列表。

```
class Post extends Model
{
    /**
     * 获得此文章的所有标签。
     */
    public function tags()
    {
        return $this->morphToMany('App\Tag', 'taggable');
    }
}
```

多对多多态关联与多对多关联的代码类似，不同的是中间表不再是两个父模型的蛇形变量，而是 `name` 的复数，值得注意的是 `foreignPivotKey` 代表中间表中对当前 `post` 或者 `video` 的外键，一般会放在 `taggable_id` 字段中，`relatedPivotKey` 代表中间表中对属性表 `tag` 的外键 `tag_id`：

```
public function morphToMany($related, $name, $table = null, $foreignPivotKey = null,
                                $relatedPivotKey = null, $parentKey = null,
                                $relatedKey = null, $inverse = false)
{
    $caller = $this->guessBelongsToManyRelation();

    $instance = $this->newRelatedInstance($related);

    $foreignPivotKey = $foreignPivotKey ?: $name.'_id';

    $relatedPivotKey = $relatedPivotKey ?: $instance->getForeignKey();

    $table = $table ?: Str::plural($name);

    return new MorphToMany(
        $instance->newQuery(), $this, $name, $table,
        $foreignPivotKey, $relatedPivotKey, $parentKey ?: $this->getKeyName(),
        $relatedKey ?: $instance->getKeyName(), $caller, $inverse
    );
}
```

`MorphToMany` 的构造函数依然有 `morphType` 与 `morphClass` ，`morphType` 标识着当前中间表的记录类型是 `Post` ，还是 `videos` ，`morphClass` 的值默认值是 `Post` 类或者 `videos` 的全名，正向关联的时候，`inverse` 是 `false` ，反向关联的时候, `inverse` 是 `true` 。

```
 public function __construct(Builder $query, Model $parent, $nam
e, $table, $foreignPivotKey,
                                $relatedPivotKey, $parentKey, $r
elatedKey, $relationName = null, $inverse = false)
{
    $this->inverse = $inverse;
    $this->morphType = $name.'_type';
    $this->morphClass = $inverse ? $query->getModel()->getMorphC
lass() : $parent->getMorphClass();

    parent::__construct(
        $query, $parent, $table, $foreignPivotKey,
        $relatedPivotKey, $parentKey, $relatedKey, $relationName
    );
}
```

正向关联的时候，`parent` 类是 `Post` 类或者 `videos` 类，反向关联的时候 `related` 是 `Post` 类或者 `videos` 类。

限制条件：

```
protected function addWhereConstraints()
{
    parent::addWhereConstraints();

    $this->query->where($this->table.'.'.$this->morphType, $this
->morphClass);

    return $this;
}

protected function addWhereConstraints()
{
    $this->query->where(
        $this->getQualifiedForeignPivotKeyName(), '=', $this->pa
rent->{$this->parentKey}
    );

    return $this;
}

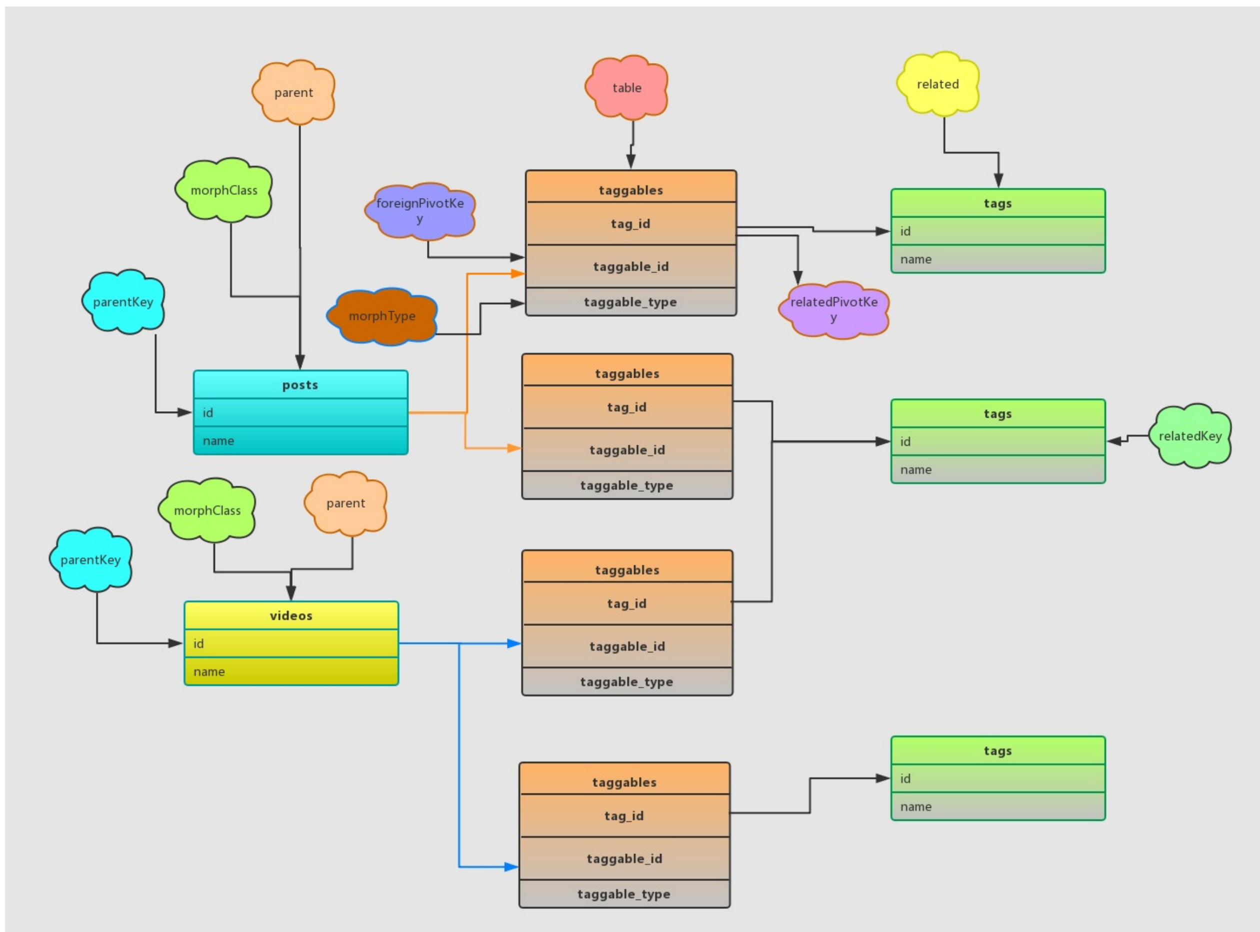
public function getQualifiedForeignPivotKeyName()
{
    return $this->table.'.'.$this->foreignPivotKey;
}
```

官网中例子限制条件转化为 `sql` (假设 `Post` 的主键为 1) ：

```
where taggables.taggable_id = 1;

where taggables.taggable_type = 'App\Post'
```

`morphToMany` 的模型关系如下：



限制条件：

```
public function addConstraints()
{
    $this->performJoin();

    if (static::$constraints) {
        $this->addWhereConstraints();
    }
}

protected function performJoin($query = null)
{
    $query = $query ?: $this->query;

    $baseTable = $this->related->getTable();

    $key = $baseTable.'.'.$this->relatedKey;

    $query->join($this->table, $key, '=', $this->getQualifiedRel
```

```
atedPivotKeyName());

    return $this;
}

protected function addWhereConstraints()
{
    parent::addWhereConstraints();

    $this->query->where($this->table.'.'.$this->morphType, $this
->morphClass);

    return $this;
}

protected function addWhereConstraints()
{
    $this->query->where(
        $this->getQualifiedForeignPivotKeyName(), '=', $this->pa
rent->{$this->parentKey}
    );

    return $this;
}
```

本例中的限制条件：

```
select tag join tagable on tagable.tag_id = tag.id

select tags where tagable.tagables_id = post.id

select tags where tagable.tagables_type = 'App\Tag'
```

多对多多态反向关联

官方文档例子：

```
class Tag extends Model
{
    /**
     * 获得此标签下所有的文章。
     */
    public function posts()
    {
        return $this->morphedByMany('App\Post', 'taggable');
    }
}
```

与正向关联相反，`relatedPivotKey` 代表中间表中对 `related` 表 `post` 或者 `video` 的外键，一般会放在 `taggable_id` 字段中，`foreignPivotKey` 代表中间表中对当前属性表 `tag` 的外键 `tag_id`：

```
public function morphedByMany($related, $name, $table = null, $f
oreignPivotKey = null,
                                  $relatedPivotKey = null, $pare
ntKey = null, $relatedKey = null)
{
    $foreignPivotKey = $foreignPivotKey ?: $this->getForeignKey(
);

    $relatedPivotKey = $relatedPivotKey ?: $name.'_id';

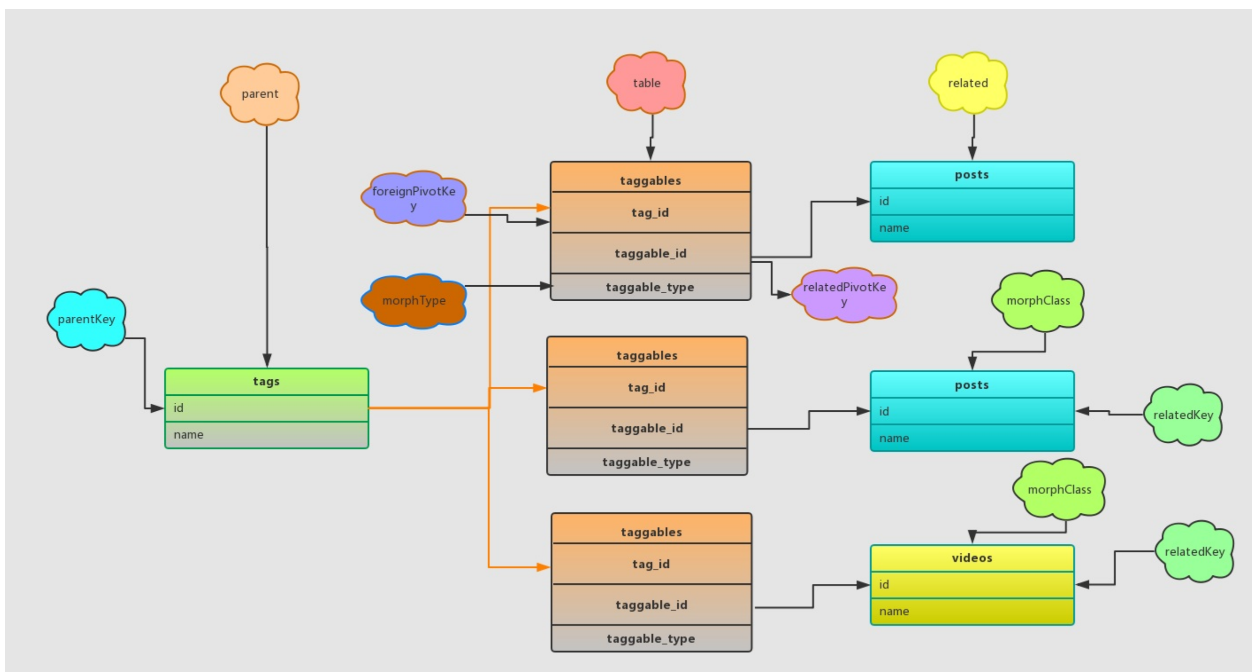
    return $this->morphToMany(
        $related, $name, $table, $foreignPivotKey,
        $relatedPivotKey, $parentKey, $relatedKey, true
    );
}
```

官网中例子限制条件转化为 `sql` (假设 `Tag` 的主键为 1)：

```
where taggables.tag_id = 1;

where taggables.taggable_type = 'App\Post'
```

`morphedByMany` 的模型关系如下：



限制条件与 `morphToMany` 一致：

```
public function addConstraints()
{
    $this->performJoin();

    if (static::$constraints) {
        $this->addWhereConstraints();
    }
}

protected function performJoin($query = null)
{
    $query = $query ?: $this->query;

    $baseTable = $this->related->getTable();

    $key = $baseTable.'.'.$this->relatedKey;

    $query->join($this->table, $key, '=', $this->getQualifiedRel
atedPivotKeyName());

    return $this;
}
```

```
protected function addWhereConstraints()
{
    parent::addWhereConstraints();

    $this->query->where($this->table.'.'.$this->morphType, $this
->morphClass);

    return $this;
}

protected function addWhereConstraints()
{
    $this->query->where(
        $this->getQualifiedForeignPivotKeyName(), '=', $this->pa
rent->{$this->parentKey}
    );

    return $this;
}
```

本例中的限制条件

```
select post join post on post.id = tagables.tagable_id

select post where tagables.tag_id = tag.id

select post where tagables.tagable_type = 'App\Post'
```

Laravel Database——Eloquent Model 模型关系加载与查询

前言

我们在上一篇文章中介绍了模型关系的定义初始化，我们可以看到，在初始化的过程中 `laravel` 已经为各种关联关系的模型预先插入了初始的 `where` 条件。本文将会进一步介绍如何添加自定义的查询条件，如何加载、预加载关联模型。

关联模型的加载

当我们定义关联模型后：

```
class User extends Model
{
    /**
     * 获得与用户关联的电话记录。
     */
    public function phone()
    {
        $this->hasOne('App\Phone', 'user_id', 'id');
    }
}
```

我们可以像成员变量一样来获取与之关联的模型：

```
$user = App\User::find(1);

foreach ($user->posts as $post) {
    //
}
```

实际上，模型的属性获取函数的确可以加载关联模型：

```
public function getAttribute($key)
{
    if (! $key) {
        return;
    }

    ...

    return $this->getRelationValue($key);
}
```

`getRelationValue` 函数专用于加载我们之前定义的关联模型：

```
public function getRelationValue($key)
{
    if ($this->relationLoaded($key)) {
        return $this->relations[$key];
    }

    if (method_exists($this, $key)) {
        return $this->getRelationshipFromMethod($key);
    }
}

public function relationLoaded($key)
{
    return array_key_exists($key, $this->relations);
}
```

可以看到，关联的加载带有缓存， `laravel` 首先会验证当前关联关系是否已经被加载，如果加载过，那么直接返回缓存结果。

```
protected function getRelationshipFromMethod($method)
{
    $relation = $this->$method();

    if (! $relation instanceof Relation) {
        throw new LogicException(get_class($this).'::'.$method.'
 must return a relationship instance.');
    }

    return tap($relation->getResults(), function ($results) use
($method) {
        $this->setRelation($method, $results);
    });
}
```

当我们调用 `$user->posts` 语句的时候，`laravel` 会调用 `posts` 函数，该函数开始定义关联关系，并且返回 `hasOne` 对象，在这里将会调用 `getResults` 函数来加载关联模型：

```
 public function getResults()
{
    return $this->query->first() ?: $this->getDefaultFor($this->
parent);
}
```

`getDefaultFor` 函数用于在未查询到任何关联模型时的情况。我们在定义关联的时候，可以提供默认的方法来控制返回的结果：

```
public function user()
{
    return $this->belongsTo('App\User')->withDefault();
}

public function user()
{
    return $this->belongsTo('App\User')->withDefault([
        'name' => '游客',
    ]);
}

public function user()
{
    return $this->belongsTo('App\User')->withDefault(function ($
user) {
        $user->name = '游客';
    });
}
```

`withDefault` 可以提供空值、数组、闭包函数等等选项，`getDefaultFor` 函数在关联没有查询到结果的时候，按要求返回一个模型：

```
public function withDefault($callback = true)
{
    $this->withDefault = $callback;

    return $this;
}

protected function getDefaultFor(Model $parent)
{
    if (! $this->withDefault) {
        return;
    }

    $instance = $this->newRelatedInstanceFor($parent);

    if (is_callable($this->withDefault)) {
        return call_user_func($this->withDefault, $instance) ?:
$instance;
    }

    if (is_array($this->withDefault)) {
        $instance->forceFill($this->withDefault);
    }

    return $instance;
}
```

获取到关联模型后，就要放入缓存当中，以备后续情况使用：

```
public function setRelation($relation, $value)
{
    $this->relations[$relation] = $value;

    return $this;
}
```

多对多关系的加载

多对多关系的加载与一对多等关系的加载有所不同，原因是不仅要加载 `related` 模型，还要加载中间表模型：

```
public function getResults()
{
    return $this->get();
}

public function get($columns = ['*'])
{
    $columns = $this->query->getQuery()->columns ? [] : $columns
;

    $builder = $this->query->applyScopes();

    $models = $builder->addSelect(
        $this->shouldSelect($columns)
    )->getModels();

    $this->hydratePivotRelation($models);

    if (count($models) > 0) {
        $models = $builder->eagerLoadRelations($models);
    }

    return $this->related->newCollection($models);
}
```

`shouldSelect` 函数加载了中间表的字段属性：

```
protected function shouldSelect(array $columns = ['*'])
{
    if ($columns == ['*']) {
        $columns = [$this->related->getTable().'.*'];
    }

    return array_merge($columns, $this->aliasedPivotColumns());
}

protected function aliasedPivotColumns()
{
    $defaults = [$this->foreignPivotKey, $this->relatedPivotKey]
;

    return collect(array_merge($defaults, $this->pivotColumns))-
>map(function ($column) {
        return $this->table.'.'.$column.' as pivot_'.$column;
    })->unique()->all();
}
```

可以看到，这个时候，中间表的属性会被放入 `related` 模型中，并且会被赋予别名前缀 `pivot_` 。

接着 `hydratePivotRelation` 会将这些中间表属性加载到中间表模型中：

```
protected function hydratePivotRelation(array $models)
{
    foreach ($models as $model) {
        $model->setRelation($this->accessor, $this->newExistingP
ivot(
            $this->migratePivotAttributes($model)
        ));
    }
}

protected function migratePivotAttributes(Model $model)
{
    $values = [];

    foreach ($model->getAttributes() as $key => $value) {
        if (strpos($key, 'pivot_') === 0) {
            $values[substr($key, 6)] = $value;

            unset($model->$key);
        }
    }

    return $values;
}
```

`accessor` 默认值为 `pivot`，我们也可以在定义多对多的时候使用 `as` 函数为它取别名：

```
return $this->belongsToMany('App\Role')->as(‘role_user’);
```

源码：

```
public function as($accessor)
{
    $this->accessor = $accessor;

    return $this;
}
```

关联模型的预加载

with 函数

当作为属性访问 Eloquent 关联时，关联数据是「懒加载」的。意味着在你第一次访问该属性时，才会加载关联数据。不过，当你查询父模型时，Eloquent 还可以进行「预加载」关联数据。预加载避免了 N + 1 查询问题。

预加载可以一次操作中预加载关联模型并且自定义用于 `select` 的列，可以预加载几个不同的关联，还可以预加载嵌套关联，预加载关联数据的时候，为查询指定额外的约束条件：

```
$books = App\Book::with(['author:id,name'])->get();

$books = App\Book::with(['author', 'publisher'])->get();

$books = App\Book::with('author.contacts')->get();

$users = App\User::with(['posts' => function ($query) {
    $query->where('title', 'like', '%first%');
}])->get();
```

我们来看看 `with` 函数：

```
public static function with($relations)
{
    return (new static)->newQuery()->with(
        is_string($relations) ? func_get_args() : $relations
    );
}
```

预加载调用 `Eloquent/builder` 的 `with` 函数：

```
public function with($relations)
{
    $eagerLoad = $this->parseWithRelations(is_string($relations)
 ? func_get_args() : $relations);

    $this->eagerLoad = array_merge($this->eagerLoad, $eagerLoad)
;

    return $this;
}
```

`eagerLoad` 成员变量用于存放预加载的关联关系，`parseWithRelations` 用于解析关联关系：

```
protected function parseWithRelations(array $relations)
{
    $results = [];

    foreach ($relations as $name => $constraints) {
        if (is_numeric($name)) {
            $name = $constraints;

            list($name, $constraints) = Str::contains($name, ':'
)
                        ? $this->createSelectWithConstraint($nam
e)
                        : [$name, function () {
                            //
                        }];
        }

        $results = $this->addNestedWiths($name, $results);

        $results[$name] = $constraints;
    }

    return $results;
}
```

当我们在模型关系中写入 `:` 符合的时候，说明我们不想 `select *`，而是想要只查询特定的字段，`createSelectWithConstraint`：

```
protected function createSelectWithConstraint($name)
{
    return [explode(':', $name)[0], function ($query) use ($name)
 {
        $query->select(explode(',', explode(':', $name)[1]));
    }];
}
```

也就是为关联关系添加 `select` 条件。

当我们想要进行嵌套查询的时候，需要在关联关系中写下 `.`，`addNestedWiths`：

```
protected function addNestedWiths($name, $results)
{
    $progress = [];

    foreach (explode('.', $name) as $segment) {
        $progress[] = $segment;

        if (! isset($results[$last = implode('.', $progress)]))
{
            $results[$last] = function () {
                //
            };
        }
    }

    return $results;
}
```

可以看到，`addNestedWiths` 为嵌套的模型关系赋予默认的空闭包函数，例如 `a.b.c`，`addNestedWiths` 返回的 `results` 数组中会有三个成员: `a`、`a.b`、`a.b.c`，这三个变量的闭包函数都是空。

接下来，`parseWithRelations` 为 `a.b.c` 的闭包函数重新赋值，将用户定义的约束条件赋予给 `a.b.c`。

get 函数预加载

`with` 函数为 `laravel` 提供了需要预加载的关联关系，`get` 函数在从数据库中获取父模型的数据后，会将需要预加载的模型也一并取出来：

```
public function get($columns = ['*'])
{
    $builder = $this->applyScopes();

    if (count($models = $builder->getModels($columns)) > 0) {
        $models = $builder->eagerLoadRelations($models);
    }

    return $builder->getModel()->newCollection($models);
}
```

顾名思义 `eagerLoadRelations` 函数就是获取预加载模型的的函数：

```
public function eagerLoadRelations(array $models)
{
    foreach ($this->eagerLoad as $name => $constraints) {
        // For nested eager loads we'll skip loading them here a
nd they will be set as an
        // eager load on the query to retrieve the relation so t
hat they will be eager
        // loaded on that query, because that is where they get
hydrated as models.
        if (strpos($name, '.') === false) {
            $models = $this->eagerLoadRelation($models, $name, $
constraints);
        }
    }

    return $models;
}
```

在这里，很让人费解的是 `if` 条件，这个条件语句看起来排除了嵌套预加载关系。例如上面的 `a.b.c`，`eagerLoadRelations` 只会加载 `a` 这个关联关系。其实原因是：

> // For nested eager loads we'll skip loading them here and they will be set as an eager load on the query to retrieve the relation so that they will be eager loaded on that query, because that is where they get hydrated as models.

翻译过来就是说，嵌套预加载要一步一步的来，第一次只加载 `a`，获得了 `a` 的关联模型之后，第二次再加载 `b`，最后加载 `c`。这里看不懂没关系，答案在下面的代码中：

```
protected function eagerLoadRelation(array $models, $name, Closure $constraints)
{
    $relation = $this->getRelation($name);

    $relation->addEagerConstraints($models);

    $constraints($relation);

    return $relation->match(
        $relation->initRelation($models, $name),
        $relation->getEager(), $name
    );
}
```

`eagerLoadRelation` 是预加载关联关系的核心，我们可以看到加载关联模型关系主要有四个步骤：

- 通过关系名来调用 `hasOne` 等函数来加载模型关系 `relation`
- 利用 `models` 来为模型关系添加约束条件
- 调用 `with` 函数附带的约束条件
- 从数据库获取关联模型并匹配到各个父模型中，作为父模型的属性

我们先从调用关联函数 `getRelation` 来说：

getRelation

```
public function getRelation($name)
{
    $relation = Relation::noConstraints(function () use ($name)
{
        try {
            return $this->getModel()->{$name}();
        } catch (BadMethodCallException $e) {
            throw RelationNotFoundException::make($this->getMode
l(), $name);
        }
    });

    $nested = $this->relationsNestedUnder($name);

    if (count($nested) > 0) {
        $relation->getQuery()->with($nested);
    }

    return $relation;
}
```

我们在上一个文章说过，`hasOne` 等函数会自动加约束条件例如：

```
select phone where phone.user_id = user.id
```

但是这个约束条件并不适用于预加载，因为预加载的父模型通常不只只一个。因此需要调用函数 `noConstraints` 来避免加载约束条件:

```
public static function noConstraints(Closure $callback)
{
    $previous = static::$constraints;

    static::$constraints = false;

    try {
        return call_user_func($callback);
    } finally {
        static::$constraints = $previous;
    }
}
```

接下来，就要调用定义关联的函数：

```
return $this->getModel()->{$name}();
```

下面的 `relationsNestedUnder` 函数用于加载嵌套的预加载关联关系：

```
protected function relationsNestedUnder($relation)
{
    $nested = [];

    foreach ($this->eagerLoad as $name => $constraints) {
        if ($this->isNestedUnder($relation, $name)) {
            $nested[substr($name, strlen($relation.'.'))] = $con
straints;
        }
    }

    return $nested;
}

protected function isNestedUnder($relation, $name)
{
    return Str::contains($name, '.') && Str::startsWith($name, $
relation.'.');
}
```

从代码上可以看出来，如果当前的模型关系是 `a`，`relationsNestedUnder` 函数会把其嵌套的关系都检测出来：`a.b`、`a.b.c`，并且放入 `nested` 数组中：`nested[b]、nested[b.c]`。

接下来：

```
if (count($nested) > 0) {
    $relation->getQuery()->with($nested);
}
```

就会继续递归预加载关联关系。

关联关系预加载约束条件

获得关联关系之后，就要加载各个关联关系自己的预加载约束条件：

```
public function addEagerConstraints(array $models)
{
    $this->query->whereIn(
        $this->foreignKey, $this->getKeys($models, $this->localK
ey)
    );
}
```

也就是从父模型的外键来为关联模型添加 `where` 条件。当然各个关联关系不同，这个函数也有一定的区别。

with 预加载约束条件

接下来还有加载 `with` 函数的约束条件 ：

```
$constraints($relation);
```

匹配父模型

当关联关系的约束条件都设置完毕后，就要从数据库中来获取关联模型：

```
$relation->match(
    $relation->initRelation($models, $name),
    $relation->getEager(), $name
);

public function getEager()
{
    return $this->get();
}
```

`initRelation` 会为父模型设置默认的关联模型：

```
public function initRelation(array $models, $relation)
{
    foreach ($models as $model) {
        $model->setRelation($relation, $this->getDefaultFor($mod
el));
    }

    return $models;
}
```

两步都做好了，接下来就要为父模型和子模型进行匹配了：

```
public function match(array $models, Collection $results, $relation)
{
    return $this->matchOne($models, $results, $relation);
}

public function matchOne(array $models, Collection $results, $relation)
{
    return $this->matchOneOrMany($models, $results, $relation, 'one');
}

protected function matchOneOrMany(array $models, Collection $results, $relation, $type)
{
    $dictionary = $this->buildDictionary($results);

    foreach ($models as $model) {
        if (isset($dictionary[$key = $model->getAttribute($this->localKey)])) {
            $model->setRelation(
                $relation, $this->getRelationValue($dictionary, $key, $type)
            );
        }
    }

    return $models;
}
```

匹配的过程分为两步：创建目录 `buildDictionary` 和设置子模型 `setRelation` ：

```
protected function buildDictionary(Collection $results)
{
    $dictionary = [];

    $foreign = $this->getForeignKeyName();

    foreach ($results as $result) {
        $dictionary[$result->{$foreign}][] = $result;
    }

    return $dictionary;
}
```

创建目录 `buildDictionary` 函数根据子模型的外键 `foreign` 将子模型进行分类，拥有同一外键的子模型放入同一个数组中。

接下来，为父模型设置子模型：

```
foreach ($models as $model) {
    if (isset($dictionary[$key = $model->getAttribute($this->loc
alKey)])) {
        $model->setRelation(
            $relation, $this->getRelationValue($dictionary, $key
, $type)
        );
    }
}

protected function getRelationValue(array $dictionary, $key, $ty
pe)
{
    $value = $dictionary[$key];

    return $type == 'one' ? reset($value) : $this->related->newC
ollection($value);
}
```

如果目录 `dictionary` 中存在父模型的主键，就会从目录中取出对应的子模型数组，并利用 `setRelation` 来为父模型设置关联模型。

关联模型的关联查询

基于存在的关联查询

官方样例：

```
// 获得所有至少有一条评论的文章...
$posts = App\Post::has('comments')->get();

// 获得所有有三条或三条以上评论的文章...
$posts = Post::has('comments', '>=', 3)->get();

// 获得所有至少有一条获赞评论的文章...
$posts = Post::has('comments.votes')->get();

// 获得所有至少有一条评论内容满足 foo% 条件的文章
$posts = Post::whereHas('comments', function ($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

`has` 函数用于基于存在的关联查询：

```
public function has($relation, $operator = '>=', $count = 1, $bo
olean = 'and', Closure $callback = null)
{
    if (strpos($relation, '.') !== false) {
        return $this->hasNested($relation, $operator, $count, $b
oolean, $callback);
    }

    $relation = $this->getRelationWithoutConstraints($relation);

    $method = $this->canUseExistsForExistenceCheck($operator, $c
ount)
                    ? 'getRelationExistenceQuery'
                    : 'getRelationExistenceCountQuery';

    $hasQuery = $relation->{$method}(
        $relation->getRelated()->newQuery(), $this
    );

    if ($callback) {
        $hasQuery->callScope($callback);
    }

    return $this->addHasWhere(
        $hasQuery, $relation, $operator, $count, $boolean
    );
}
```

`has` 函数的步骤：

- 获取无约束的关联关系
- 为关联关系添加 `existence` 约束
- 为关联关系添加 `has` 外部约束
- 将关联关系添加到 `where` 条件中

无约束的关联关系

```
protected function getRelationWithoutConstraints($relation)
{
    return Relation::noConstraints(function () use ($relation) {
        return $this->getModel()->{$relation}();
    });
}
```

这个不用多说，和预加载的原理一样。

existence 约束

关系模型的 `existence` 约束条件很简单：

```
select * from post where user.id = post.user_id
```

`laravel` 还考虑一种特殊情况，那就是自己关联自己，这个时候就会为模型命名一个新的 `hash` ：

```
select * from user as wedfklk where user.id = wedfklk.foreignKey
```

源代码比较简单：

```
public function getRelationExistenceQuery(Builder $query, Builde
r $parentQuery, $columns = ['*'])
{
    if ($query->getQuery()->from == $parentQuery->getQuery()->fr
om) {
        return $this->getRelationExistenceQueryForSelfRelation($
query, $parentQuery, $columns);
    }

    return parent::getRelationExistenceQuery($query, $parentQuer
y, $columns);
}

public function getRelationExistenceQueryForSelfRelation(Builder
 $query, Builder $parentQuery, $columns = ['*'])
```

```
{
    $query->from($query->getModel()->getTable().' as '.$hash = $
this->getRelationCountHash());

    $query->getModel()->setTable($hash);

    return $query->select($columns)->whereColumn(
        $this->getQualifiedParentKeyName(), '=', $hash.'.'.$this
->getForeignKeyName()
    );
}

public function getRelationExistenceQuery(Builder $query, Builde
r $parentQuery, $columns = ['*'])
{
    return $query->select($columns)->whereColumn(
        $this->getQualifiedParentKeyName(), '=', $this->getExist
enceCompareKey()
    );
}

public function getExistenceCompareKey()
{
    return $this->getQualifiedForeignKeyName();
}
```

ExistenceCount 约束

`ExistenceCount` 约束只是 `select *` 变成了 `select count(*)`：

```
select count(*) from post where user.id = post.user_id
```

源代码：

```
public function getRelationExistenceCountQuery(Builder $query, Builder $parentQuery)
{
    return $this->getRelationExistenceQuery(
        $query, $parentQuery, new Expression('count(*)')
    );
}
```

关联关系添加到 `where` 条件

当关联关系的 `存在` 约束设置完毕后，就要加载到父模型的 `where` 条件中，一般会作为父模型的子查询：

```
protected function addHasWhere(Builder $hasQuery, Relation $relation, $operator, $count, $boolean)
{
    $hasQuery->mergeConstraintsFrom($relation->getQuery());

    return $this->canUseExistsForExistenceCheck($operator, $count)
            ? $this->addWhereExistsQuery($hasQuery->toBase(), $boolean, $operator === '<' && $count === 1)
            : $this->addWhereCountQuery($hasQuery->toBase(), $operator, $count, $boolean);
}

public function addWhereExistsQuery(Builder $query, $boolean = 'and', $not = false)
{
    $type = $not ? 'NotExists' : 'Exists';

    $this->wheres[] = compact('type', 'operator', 'query', 'boolean');

    $this->addBinding($query->getBindings(), 'where');

    return $this;
}

protected function addWhereCountQuery(QueryBuilder $query, $operator = '>=', $count = 1, $boolean = 'and')
{
    $this->query->addBinding($query->getBindings(), 'where');

    return $this->where(
        new Expression('('.$query->toSql().')'),
        $operator,
        is_numeric($count) ? new Expression($count) : $count,
        $boolean
    );
}
```

`existence` 约束最后条件：

```
select * from user where exists (select * from phone where phone
.user_id=user.id)
```

`ExistenceCount` 约束:

```
select * from user where (select count(*) from phone where phone
.user_id=user.id) >= 3
```

嵌套查询

嵌套查询需要进行递归来调用 `has` 函数：

```
protected function hasNested($relations, $operator = '>=', $coun
t = 1, $boolean = 'and', $callback = null)
{
    $relations = explode('.', $relations);

    $closure = function ($q) use (&$closure, &$relations, $opera
tor, $count, $callback) {
        count($relations) > 1
            ? $q->whereHas(array_shift($relations), $closure)
            : $q->has(array_shift($relations), $operator, $count
, 'and', $callback);
    };

    return $this->has(array_shift($relations), '>=', 1, $boolean
, $closure);
}

public function whereHas($relation, Closure $callback = null, $o
perator = '>=', $count = 1)
{
    return $this->has($relation, $operator, $count, 'and', $call
back);
}
```

例如

```
$posts = Post::has('comments.votes')->get();
```

首先 `hasNested` 会返回：

```
$this->has('comments', '>=', 1, 'and', function ($q) use (&$closure, ‘votes’, '>=', 1, $callback) {
        $q->has(‘votes’, '>=', 1, 'and', $callback);
    }
);
```

生成的 sql:

```
select * from post where exist (select * from comment where comment.post_id=post.id and where exist (select * from vote where vote.comment_id=comment.id))
```

基于不存在的关联查询

基于不存在的关联查询只是基于存在的关联查询

```
public function doesntHave($relation, $boolean = 'and', Closure $callback = null)
{
    return $this->has($relation, '<', 1, $boolean, $callback);
}

public function whereDoesntHave($relation, Closure $callback = null)
{
    return $this->doesntHave($relation, 'and', $callback);
}
```

关联数据计数

如果您只想统计结果数而不需要加载实际数据，那么可以使用 withCount 方法，此方法会在您的结果集模型中添加一个 {关联名}_count 字段。例如：

```
$posts = App\Post::withCount('comments')->get();
//select *,(select count(*) from comment where comment.post_id=p
ost.id) as comments_count from post

foreach ($posts as $post) {
    echo $post->comments_count;
}

//多个关联数据「计数」，并为其查询添加约束条件：
$posts = Post::withCount(['votes', 'comments' => function ($quer
y) {
    $query->where('content', 'like', 'foo%');
}])->get();
//select *,(select count(*) from comment where comment.post_id=p
ost.id and content like 'foo%') as comments_count,(select count(
*) from votes where vote.post_id=post.id) as votes_count from po
st

echo $posts[0]->votes_count;
echo $posts[0]->comments_count;

//可以为关联数据计数结果起别名，允许在同一个关联上多次计数：
$posts = Post::withCount([
    'comments',
    'comments as pending_comments_count' => function ($query) {
        $query->where('approved', false);
    }
])->get();
//select *,(select count(*) from comment where comment.post_id=p
ost.id) as comments_count,(select count(*) from comment where co
mment.post_id=post.id and approved=false) as pending_comments_co
unt from post

echo $posts[0]->comments_count;

echo $posts[0]->pending_comments_count;
```

`withCount` 的源代码与 `has` 的代码高度相似：

```
public function withCount($relations)
{
    if (empty($relations)) {
        return $this;
    }

    if (is_null($this->query->columns)) {
        $this->query->select([$this->query->from.'.*']);
    }

    $relations = is_array($relations) ? $relations : func_get_ar
gs();

    foreach ($this->parseWithRelations($relations) as $name => $
constraints) {
        $segments = explode(' ', $name);

        unset($alias);

        if (count($segments) == 3 && Str::lower($segments[1]) ==
'as') {
            list($name, $alias) = [$segments[0], $segments[2]];
        }

        $relation = $this->getRelationWithoutConstraints($name);

        $query = $relation->getRelationExistenceCountQuery(
            $relation->getRelated()->newQuery(), $this
        );

        $query->callScope($constraints);

        $query->mergeConstraintsFrom($relation->getQuery());

        $column = $alias ?? Str::snake($name.'_count');

        $this->selectSub($query->toBase(), $column);
    }
```

```
    return $this;
}
```

- 解析关联关系名称
- 获取无约束的关联关系
- 为关联关系添加 `existenceCount` 约束
- 为关联关系添加 `with` 外部约束
- 将关联关系添加到 `where` 条件中
- 设置 `alias` 别名
- 创建 `select` 子查询

多对多关系的中间表查询

```
return $this->belongsToMany('App\Role')->wherePivot('approved', 1
);

return $this->belongsToMany('App\Role')->wherePivotIn('priority'
, [1, 2]);
```

```
public function wherePivot($column, $operator = null, $value = n
ull, $boolean = 'and')
{
    $this->pivotWheres[] = func_get_args();

    return $this->where($this->table.'.'.$column, $operator, $va
lue, $boolean);
}

public function wherePivotIn($column, $values, $boolean = 'and',
 $not = false)
{
    $this->pivotWhereIns[] = func_get_args();

    return $this->whereIn($this->table.'.'.$column, $values, $bo
olean, $not);
}
```

注意这里的 `pivotWheres` 与 `pivotWheres` 变量，这个变量在对中间表的加载中会被使用：

```
protected function newPivotQuery()
{
    $query = $this->newPivotStatement();

    foreach ($this->pivotWheres as $arguments) {
        call_user_func_array([$query, 'where'], $arguments);
    }

    foreach ($this->pivotWhereIns as $arguments) {
        call_user_func_array([$query, 'whereIn'], $arguments);
    }

    return $query->where($this->foreignPivotKey, $this->parent->
{$this->parentKey});
}
```

Laravel Database——Eloquent Model 更新关联模型

前言

在前两篇文章中，向大家介绍了定义关联关系的源码，还有基于关联关系的关联模型加载与查询的源码分析，本文开始介绍第三部分，如何利用关联关系来更新插入关联模型。

hasOne/hasMany/MorphOne/MorphMany 更新与插入

save 方法

正向的一对一、一对多关联保存方法用于对子模型设置外键值：

```
public function save(Model $model)
{
    $this->setForeignAttributesForCreate($model);

    return $model->save() ? $model : false;
}

protected function setForeignAttributesForCreate(Model $model)
{
    $model->setAttribute($this->getForeignKeyName(), $this->getP
arentKey());
}

public function getParentKey()
{
    return $this->parent->getAttribute($this->localKey);
}
```

saveMany 方法

```
public function saveMany($models)
{
    foreach ($models as $model) {
        $this->save($model);
    }

    return $models;
}
```

create 方法

`create` 方法与 `save` 方法功能一致，唯一不同的是 `create` 的参数是属性， `save` 方法的参数是 `model` 。

```
public function create(array $attributes = [])
{
    return tap($this->related->newInstance($attributes), function
 ($instance) {
        $this->setForeignAttributesForCreate($instance);

        $instance->save();
    });
}

protected function setForeignAttributesForCreate(Model $model)
{
    $model->setAttribute($this->getForeignKeyName(), $this->getP
arentKey());
}
```

createMany 方法

```
public function createMany(array $records)
{
    $instances = $this->related->newCollection();

    foreach ($records as $record) {
        $instances->push($this->create($record));
    }

    return $instances;
}
```

make 方法

`make` 方法用于建立子模型对象，但是并不进行保存操作：

```
public function make(array $attributes = [])
{
    return tap($this->related->newInstance($attributes), function
 ($instance) {
        $this->setForeignAttributesForCreate($instance);
    });
}
```

update 方法

`update` 方法用于更新子模型的属性，值得注意的是时间戳的更新：

```
public function update(array $attributes)
{
    if ($this->related->usesTimestamps()) {
        $attributes[$this->relatedUpdatedAt()] = $this->related-
>freshTimestampString();
    }

    return $this->query->update($attributes);
}
```

findOrNew 方法

```
public function findOrNew($id, $columns = ['*'])
{
    if (is_null($instance = $this->find($id, $columns))) {
        $instance = $this->related->newInstance();

        $this->setForeignAttributesForCreate($instance);
    }

    return $instance;
}
```

firstOrCreate 方法

实际调用的是 `create` 方法：

```
public function firstOrCreate(array $attributes, array $values =
 [])
{
    if (is_null($instance = $this->where($attributes)->first()))
 {
        $instance = $this->create($attributes + $values);
    }

    return $instance;
}
```

updateOrCreate 方法

```
public function updateOrCreate(array $attributes, array $values
= [])
{
    return tap($this->firstOrNew($attributes), function ($instan
ce) use ($values) {
        $instance->fill($values);

        $instance->save();
    });
}
```

belongsTo/MorphTo 更新

save 方法

如果我们在子模型加一个包含关联名称的 `touches` 属性后，当我们更新一个子模型时，对应父模型的 `updated_at` 字段也会被同时更新:

```
class Comment extends Model
{
    protected $touches = ['post'];

    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}


$comment = App\Comment::find(1);

$comment->text = '编辑了这条评论！';

$comment->save();
```

这是由于，对子模型调用 `save` 方法会引发 `finishSave` 函数：

```
protected function finishSave(array $options)
{
    $this->fireModelEvent('saved', false);

    if ($this->isDirty() && ($options['touch'] ?? true)) {
        $this->touchOwners();
    }

    $this->syncOriginal();
}
```

可以看到， `touchOwners` 函数被调用：

```
public function touchOwners()
{
    foreach ($this->touches as $relation) {
        $this->$relation()->touch();

        if ($this->$relation instanceof self) {
            $this->$relation->fireModelEvent('saved', false);

            $this->$relation->touchOwners();
        } elseif ($this->$relation instanceof Collection) {
            $this->$relation->each(function (Model $relation) {
                $relation->touchOwners();
            });
        }
    }
}
```

可以看到， `touchOwners` 函数会调用 `touch` 函数，该函数用于更新父模型的时间戳：

```
public function touch()
{
    $column = $this->getRelated()->getUpdatedAtColumn();

    $this->rawUpdate([$column => $this->getRelated()->freshTimes
tampString()]);
}
```

之后，父模型还会递归调用 `touchOwners` 函数，不断更新上一级的父模型。

update 方法

`belongsTo/MorphTo` 的更新方法用于父模型的属性更新：

```
public function update(array $attributes)
{
    return $this->getResults()->fill($attributes)->save();
}
```

associate 方法

如果想要更新 `belongsTo` 关联时，可以使用 `associate` 方法。此方法会在子模型中设置外键：

```
public function associate($model)
{
    $ownerKey = $model instanceof Model ? $model->getAttribute($
this->ownerKey) : $model;

    $this->child->setAttribute($this->foreignKey, $ownerKey);

    if ($model instanceof Model) {
        $this->child->setRelation($this->relation, $model);
    }

    return $this->child;
}
```

dissociate 方法

当删除 belongsTo 关联时，可以使用 dissociate方法。此方法会设置关联外键为 null：

```
public function dissociate()
{
    $this->child->setAttribute($this->foreignKey, null);

    return $this->child->setRelation($this->relation, null);
}
```

belongsToMany/MorphToMany/MorphByMany 更新与插入

attach 方法

`attach` 方法用于为多对多关系添加新的关联关系，主要进行了中间表的插入工作，用法：

```
$user = App\User::find(1);

$user->roles()->attach($roleId);

//也可以通过传递一个数组参数向中间表写入额外数据
$user->roles()->attach($roleId, ['expires' => $expires]);

//为了方便，还允许传入 ID 数组：
$user->roles()->attach([
    1 => ['expires' => $expires],
    2 => ['expires' => $expires]
]);
```

源码：

```
public function attach($id, array $attributes = [], $touch = tru
e)
{
    $this->newPivotStatement()->insert($this->formatAttachRecord
s(
        $this->parseIds($id), $attributes
    ));

    if ($touch) {
        $this->touchIfTouching();
    }
}

protected function parseIds($value)
{
    if ($value instanceof Model) {
        return [$value->getKey()];
    }

    if ($value instanceof Collection) {
        return $value->modelKeys();
    }

    if ($value instanceof BaseCollection) {
        return $value->toArray();
    }

    return (array) $value;
}

public function newPivotStatement()
{
    return $this->query->getQuery()->newQuery()->from($this->tab
le);
}
```

可以看到，`attach` 函数最重要的是对中间表插入新数据。

在说这段代码之前，我们要先说说多对多关联关系独有的设置：

中间表 Pivot 特殊初始化设置

- 自定义中间表模型

```
class Role extends Model
{
    /**
     * 获得此角色下的用户。
     */
    public function users()
    {
        return $this->belongsToMany('App\User')->using('App\User
Role');
    }
}
```

`using` 源码非常简单：

```
public function using($class)
{
    $this->using = $class;

    return $this;
}
```

- 中间表时间戳字段

```
return $this->belongsToMany('App\Role')->withTimestamps();
```

`withTimestamps` 源码：

```
public function withTimestamps($createdAt = null, $updatedAt = null)
{
    $this->pivotCreatedAt = $createdAt;
    $this->pivotUpdatedAt = $updatedAt;

    return $this->withPivot($this->createdAt(), $this->updatedAt());
}

public function createdAt()
{
    return $this->pivotCreatedAt ?: $this->parent->getCreatedAtColumn();
}

public function updatedAt()
{
    return $this->pivotUpdatedAt ?: $this->parent->getUpdatedAtColumn();
}
```

- 中间表自定义字段

```
return $this->belongsToMany('App\Role')->withPivot('column1', 'column2');
```

自定义字段都会存放在 `pivotColumns` 中：

```
public function withPivot($columns)
{
    $this->pivotColumns = array_merge(
        $this->pivotColumns, is_array($columns) ? $columns : fun
c_get_args()
    );

    return $this;
}
```

中间表时间戳

我们接着说中间表的插入代码：

```
protected function formatAttachRecords($ids, array $attributes)
{
    $records = [];

    $hasTimestamps = ($this->hasPivotColumn($this->createdAt())
||
              $this->hasPivotColumn($this->updatedAt()));

    $attributes = $this->using
            ? $this->newPivot()->forceFill($attributes)->getAttr
ibutes()
            : $attributes;

    foreach ($ids as $key => $value) {
        $records[] = $this->formatAttachRecord(
            $key, $value, $attributes, $hasTimestamps
        );
    }

    return $records;
}
```

如果我们在设置多对多关联关系的时候，使用了时间戳，那么 `hasTimestamps` 就会为 `true` 。

初始化 Pivot

当我们设置了自定义的中间表模型时，就会调用 `newPivot` 函数：

```
public function newPivot(array $attributes = [], $exists = false)

{
    $pivot = $this->related->newPivot(
        $this->parent, $attributes, $this->table, $exists, $this
->using
    );

    return $pivot->setPivotKeys($this->foreignPivotKey, $this->r
elatedPivotKey);
}

public function newPivot(Model $parent, array $attributes, $tabl
e, $exists, $using = null)
{
    return $using ? $using::fromRawAttributes($parent, $attribut
es, $table, $exists)
                  : Pivot::fromAttributes($parent, $attributes,
$table, $exists);
}

public function setPivotKeys($foreignKey, $relatedKey)
{
    $this->foreignKey = $foreignKey;

    $this->relatedKey = $relatedKey;

    return $this;
}
```

可以看到，`newPivot` 会返回 `Pivot` 类型的对象，另外为中间表设置了 `foreignKey` 与 `relatedKey`

生成 insert 数组

```
protected function formatAttachRecord($key, $value, $attributes,
 $hasTimestamps)
{
    list($id, $attributes) = $this->extractAttachIdAndAttributes
($key, $value, $attributes);

    return array_merge(
        $this->baseAttachRecord($id, $hasTimestamps), $attribute
s
    );
}

protected function extractAttachIdAndAttributes($key, $value, ar
ray $attributes)
{
    return is_array($value)
                ? [$key, array_merge($value, $attributes)]
                : [$value, $attributes];
}
```

`extractAttachIdAndAttributes` 用于获得插入记录的主键 `id`，与其对应的属性。由于可以这样进行传入参数：

```
$user->roles()->attach([
    1 => ['expires' => $expires],
    2 => ['expires' => $expires]
]);
```

所以要判断一下 `value` 是否是数组。`baseAttachRecord` 最终生成用于 `insert` 的属性数组：

```
protected function baseAttachRecord($id, $timed)
{
    $record[$this->relatedPivotKey] = $id;

    $record[$this->foreignPivotKey] = $this->parent->{$this->par
entKey};

    if ($timed) {
        $record = $this->addTimestampsToAttachment($record);
    }

    return $record;
}

protected function addTimestampsToAttachment(array $record, $exi
sts = false)
{
    $fresh = $this->parent->freshTimestamp();

    if (! $exists && $this->hasPivotColumn($this->createdAt()))
{
        $record[$this->createdAt()] = $fresh;
    }

    if ($this->hasPivotColumn($this->updatedAt())) {
        $record[$this->updatedAt()] = $fresh;
    }

    return $record;
}
```

touchIfTouching 更新多对多时间戳更新

对中间表进行插入操作后，就要对父模型与 `related` 模型进行时间戳更新操作：

```
public function touchIfTouching()
{
    if ($this->touchingParent()) {
        $this->getParent()->touch();
    }

    if ($this->getParent()->touches($this->relationName)) {
        $this->touch();
    }
}

public function touch()
{
    if (! $this->usesTimestamps()) {
        return false;
    }

    $this->updateTimestamps();

    return $this->save();
}
```

首先，如果 `related` 模型的 `touchs` 数组中有本多对多关系，那么父模型就要进行时间戳更新操作：

```
protected function touchingParent()
{
    return $this->getRelated()->touches($this->guessInverseRelat
ion());
}

protected function guessInverseRelation()
{
    return Str::camel(Str::plural(class_basename($this->getParen
t())));
}
```

其次，如果父模型的 `touchs` 数组中存在多对多关联，那么就要进行多对多关联的 `touch` 函数，对 `related` 模型进行时间戳更新操作：

```
public function touch()
{
    $key = $this->getRelated()->getKeyName();

    $columns = [
        $this->related->getUpdatedAtColumn() => $this->related->
freshTimestampString(),
    ];

    if (count($ids = $this->allRelatedIds()) > 0) {
        $this->getRelated()->newQuery()->whereIn($key, $ids)->up
date($columns);
    }
}

public function allRelatedIds()
{
    return $this->newPivotQuery()->pluck($this->relatedPivotKey)
;
}
```

save 方法

`belongsToMany` 的 `save` 方法用于更新多对多关系，该函数会：

- 更新 `related` 模型属性
- 在中间表中添加新的记录
- 更新父模型与 `related` 模型的时间戳

主要调用了 `attach` 函数：

```
public function save(Model $model, array $pivotAttributes = [],
$touch = true)
{
    $model->save(['touch' => false]);

    $this->attach($model->getKey(), $pivotAttributes, $touch);

    return $model;
}
```

saveMany 方法

```
public function saveMany($models, array $pivotAttributes = [])
{
    foreach ($models as $key => $model) {
        $this->save($model, (array) ($pivotAttributes[$key] ?? [
]), false);
    }

    $this->touchIfTouching();

    return $models;
}
```

create 方法

多对多的 `create` 方法用于保存 `related` 的属性，并且可以为中间表添加 `joining` 属性信息：

```
public function create(array $attributes = [], array $joining =
[], $touch = true)
{
    $instance = $this->related->newInstance($attributes);

    $instance->save(['touch' => false]);

    $this->attach($instance->getKey(), $joining, $touch);

    return $instance;
}
```

createMany 方法

```
public function createMany(array $records, array $joinings = [])
{
    $instances = [];

    foreach ($records as $key => $record) {
        $instances[] = $this->create($record, (array) ($joinings
[$key] ?? []), false);
    }

    $this->touchIfTouching();

    return $instances;
}
```

detach 方法

`detach` 方法比较简单，重要的是对中间表进行删除操作：

```
public function detach($ids = null, $touch = true)
{
    $query = $this->newPivotQuery();

    if (! is_null($ids)) {
        $ids = $this->parseIds($ids);

        if (empty($ids)) {
            return 0;
        }

        $query->whereIn($this->relatedPivotKey, (array) $ids);
    }

    $results = $query->delete();

    if ($touch) {
        $this->touchIfTouching();
    }

    return $results;
}
```

同步关联 sync

```
$user->roles()->sync([1, 2, 3]);

//可以通过 ID 传递其他额外的数据到中间表：
$user->roles()->sync([1 => ['expires' => true], 2, 3]);
```

源码：

```
public function sync($ids, $detaching = true)
{
    $changes = [
        'attached' => [], 'detached' => [], 'updated' => [],
    ];

    $current = $this->newPivotQuery()->pluck(
        $this->relatedPivotKey
    )->all();

    $detach = array_diff($current, array_keys(
        $records = $this->formatRecordsList($this->parseIds($ids
))
    ));

    if ($detaching && count($detach) > 0) {
        $this->detach($detach);

        $changes['detached'] = $this->castKeys($detach);
    }

    $changes = array_merge(
        $changes, $this->attachNew($records, $current, false)
    );

    if (count($changes['attached']) ||
        count($changes['updated'])) {
        $this->touchIfTouching();
    }

    return $changes;
}
```

同步关联需要删除未出现的 `id`，更新已经存在 `id`，增添新出现的 `id`。

```
 $current = $this->newPivotQuery()->pluck(
    $this->relatedPivotKey
)->all();
```

这句用于从中间表中取出所有关联的中间表记录，并且取出 `relatedPivotKey` 值。

```
$detach = array_diff($current, array_keys(
    $records = $this->formatRecordsList($this->parseIds($ids))
));

protected function formatRecordsList(array $records)
{
    return collect($records)->mapWithKeys(function ($attributes,
 $id) {
        if (! is_array($attributes)) {
            list($id, $attributes) = [$attributes, []];
        }

        return [$id => $attributes];
    })->all();
}
```

这句用于统计出待删除的中间表记录的 `relatedPivotKey` 值。

```
if ($detaching && count($detach) > 0) {
    $this->detach($detach);

    $changes['detached'] = $this->castKeys($detach);
}
```

这句进行删除操作。

```
$changes = array_merge(
    $changes, $this->attachNew($records, $current, false)
);

protected function attachNew(array $records, array $current, $to
uch = true)
{
    $changes = ['attached' => [], 'updated' => []];

    foreach ($records as $id => $attributes) {
        if (! in_array($id, $current)) {
            $this->attach($id, $attributes, $touch);

            $changes['attached'][] = $this->castKey($id);
        }

        elseif (count($attributes) > 0 &&
            $this->updateExistingPivot($id, $attributes, $touch)
) {
            $changes['updated'][] = $this->castKey($id);
        }
    }

    return $changes;
}
```

对于需要新增的记录，直接调用方法 `attach` 即可。对于需要更新的记录，需要调用 `updateExistingPivot` :

```
public function updateExistingPivot($id, array $attributes, $tou
ch = true)
{
    if (in_array($this->updatedAt(), $this->pivotColumns)) {
        $attributes = $this->addTimestampsToAttachment($attribut
es, true);
    }

    $updated = $this->newPivotStatementForId($id)->update($attri
butes);

    if ($touch) {
        $this->touchIfTouching();
    }

    return $updated;
}

public function newPivotStatementForId($id)
{
    return $this->newPivotQuery()->where($this->relatedPivotKey,
 $id);
}
```

这个函数主要调用 `update` 方法。

切换关联 toggle

多对多关联也提供了一个 toggle 方法用于「切换」给定 IDs 的附加状态。如果给定 ID 已附加，就会被移除。同样的，如果给定 ID 已移除，就会被附加，源码：

```
public function toggle($ids, $touch = true)
{
    $changes = [
        'attached' => [], 'detached' => [],
    ];

    $records = $this->formatRecordsList($this->parseIds($ids));

    $detach = array_values(array_intersect(
        $this->newPivotQuery()->pluck($this->relatedPivotKey)->a
ll(),
        array_keys($records)
    ));

    if (count($detach) > 0) {
        $this->detach($detach, false);

        $changes['detached'] = $this->castKeys($detach);
    }

    $attach = array_diff_key($records, array_flip($detach));

    if (count($attach) > 0) {
        $this->attach($attach, [], false);

        $changes['attached'] = array_keys($attach);
    }

    if ($touch && (count($changes['attached']) ||
                   count($changes['detached']))) {
        $this->touchIfTouching();
    }

    return $changes;
}
```

`toggle` 函数先 `intersect` 被关联的主键，进行 `detach` 所有已经存在的记录，再 `diff` 被关联的主键，对其进行 `attach` 所有记录。

findOrNew 方法

`findOrNew` 函数用于 `related` 模型的主键搜索与新建：

```
public function findOrNew($id, $columns = ['*'])
{
    if (is_null($instance = $this->find($id, $columns))) {
        $instance = $this->related->newInstance();
    }

    return $instance;
}
```

firstOrNew 方法

`firstOrNew` 函数用于 `related` 模型的属性搜索与新建：

```
public function firstOrNew(array $attributes)
{
    if (is_null($instance = $this->where($attributes)->first()))
 {
        $instance = $this->related->newInstance($attributes);
    }

    return $instance;
}
```

firstOrCreate 方法

`firstOrCreate` 函数用于 `related` 模型的属性搜索与保存，`attributes` 是 `related` 模型的搜索属性或保存属性，`joining` 是中间表属性：

```
public function firstOrCreate(array $attributes, array $joining
= [], $touch = true)
{
    if (is_null($instance = $this->where($attributes)->first()))
 {
        $instance = $this->create($attributes, $joining, $touch)
;
    }

    return $instance;
}
```

updateOrCreate 方法

`updateOrCreate` 函数用于 `related` 模型的更新，`attributes` 是 `related` 模型的搜索属性，`values` 是 `related` 模型的更新属性，`joining` 是中间表属性：

```
public function updateOrCreate(array $attributes, array $values
= [], array $joining = [], $touch = true)
{
    if (is_null($instance = $this->where($attributes)->first()))
 {
        return $this->create($values, $joining, $touch);
    }

    $instance->fill($values);

    $instance->save(['touch' => false]);

    return $instance;
}
```

Laravel Session——session 的启动与运行源码分析

前言

在网页开发中，`session` 具有重要的作用，它可以在多个请求中存储用户的信息，用于识别用户的身份信息。`laravel` 为用户提供了可读性强的 API 处理各种自带的 Session 后台驱动程序。支持诸如比较热门的 Memcached、Redis 和开箱即用的数据库等常见的后台驱动程序。本文将会在本篇文章中讲述最常见的由 `File` 与 `redis` 驱动的 `session` 源码。

session 服务的注册

与其他功能一样，`session` 由自己的服务提供者在 `container` 内进行注册：

```
class SessionServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->registerSessionManager();

        $this->registerSessionDriver();

        $this->app->singleton(StartSession::class);
    }

    protected function registerSessionManager()
    {
        $this->app->singleton('session', function ($app) {
            return new SessionManager($app);
        });
    }

    protected function registerSessionDriver()
    {
        $this->app->singleton('session.store', function ($app) {
            return $app->make('session')->driver();
        });
    }

}
```

可以看到 `SessionManager` 是整个 `session` 服务的接口类，一切对 `session` 的操作都是由这个类实现。`session.store` 是 `session` 服务的存储驱动。

session 服务的启动

`session` 服务是以中间件的形式启动的，其中间件是 `Illuminate\Session\Middleware\StartSession`：

```
public function handle($request, Closure $next)
{
    $this->sessionHandled = true;

    if ($this->sessionConfigured()) {
        $request->setLaravelSession(
            $session = $this->startSession($request)
        );

        $this->collectGarbage($session);
    }

    $response = $next($request);

    if ($this->sessionConfigured()) {
        $this->storeCurrentUrl($request, $session);

        $this->addCookieToResponse($response, $session);
    }

    return $response;
}

public function terminate($request, $response)
{
    if ($this->sessionHandled && $this->sessionConfigured() && !
 $this->usingCookieSessions()) {
        $this->manager->driver()->save();
    }
}
```

`session` 服务的中间件在 `http` 会话前与会话后都有处理。

在会话前，

- `laravel` 试图从 `cookies` 中获取 `sessionId` ；
- 利用 `sessionId` 读取服务器中的 `session` 数据；
- 将 `session` 对象存入 `request` 中；
- `session` 垃圾回收

在会话后，

- 存储当前的 `url` 作为 `session` 的 `PreviousUrl`
- 将当前的 `session` 存入浏览器 `cookies` 中
- 保存当前的 `session` 数据到存储器驱动

startSession

`startSession` 函数进行了 `session` 的启动工作：

```
public function __construct(SessionManager $manager)
{
    $this->manager = $manager;
}

protected function startSession(Request $request)
{
    return tap($this->getSession($request), function ($session)
use ($request) {
        $session->setRequestOnHandler($request);

        $session->start();
    });
}

public function getSession(Request $request)
{
    return tap($this->manager->driver(), function ($session) use
($request) {
        $session->setId($request->cookies->get($session->getName
()));
    });
}
```

session 的门面类 sessionManager

代码很简洁，`session` 服务启动的逻辑被包含在了 `sessionManager` 中，`sessionManager` 是 `session` 服务的门面类，负责 `session` 服务的驱动加载与数据操作。

首先我们先看看 `SessionManager`:

```
namespace Illuminate\Session;

use Illuminate\Support\Manager;

class SessionManager extends Manager
{

}
```

`SessionManager` 继承 `Manager` 类：

```
namespace Illuminate\Support;

abstract class Manager
{
    public function driver($driver = null)
    {
        $driver = $driver ?: $this->getDefaultDriver();

        if (! isset($this->drivers[$driver])) {
            $this->drivers[$driver] = $this->createDriver($drive
r);
        }

        return $this->drivers[$driver];
    }
}
```

当我们调用 `driver` 函数的时候，程序就开始为 `session` 服务加载驱动，例如对数据库或者 `redis` 驱动，进行 `连接` 操作。

```
public function getDefaultDriver()
{
    return $this->app['config']['session.driver'];
}

protected function createDriver($driver)
{
    $method = 'create'.Str::studly($driver).'Driver';

    if (isset($this->customCreators[$driver])) {
        return $this->callCustomCreator($driver);
    } elseif (method_exists($this, $method)) {
        return $this->$method();
    }

    throw new InvalidArgumentException("Driver [$driver] not sup
ported.");
}
```

session 驱动持久化类 SessionHandler

`FileSessionHandler` 这个类就是驱动，它继承 `SessionHandlerInterface` 基类，任何对 `session` 的读取、添加、删除、更新等等操作最后都要通过这个驱动类进行持久化。

- `file` 驱动：

`file` 驱动的核心是 `Filesystem` ，该类是 Ioc 容器创建的：

```
protected function createFileDriver()
{
    return $this->createNativeDriver();
}

protected function createNativeDriver()
{
    $lifetime = $this->app['config']['session.lifetime'];

    return $this->buildSession(new FileSessionHandler(
        $this->app['files'], $this->app['config']['session.files'
], $lifetime
    ));
}

namespace Illuminate\Session;

class FileSessionHandler implements SessionHandlerInterface
{
    public function __construct(Filesystem $files, $path, $minut
es)
    {
        $this->path = $path;
        $this->files = $files;
        $this->minutes = $minutes;
    }
}
```

- redis 驱动

`redis` 驱动并不是直接创建 `redis` ，而是利用了 `laravel` 的缓存 `cache` 系统创建 `redis` 驱动，然后对 `redis` 驱动进行连接操作：

```
protected function createRedisDriver()
{
    $handler = $this->createCacheHandler('redis');

    $handler->getCache()->getStore()->setConnection(
        $this->app['config']['session.connection']
    );

    return $this->buildSession($handler);
}


protected function createCacheHandler($driver)
{
    $store = $this->app['config']->get('session.store') ?: $driv
er;

    return new CacheBasedSessionHandler(
        clone $this->app['cache']->store($store),
        $this->app['config']['session.lifetime']
    );
}

class CacheBasedSessionHandler implements SessionHandlerInterface

{
    public function __construct(CacheContract $cache, $minutes)
    {
        $this->cache = $cache;
        $this->minutes = $minutes;
    }
}
```

session 数据操作类

`buildSession` 函数将会返回 `Store` 类，这个 `Store` 类实际上 `session` 服务数据操作的实质类，任何对 `session` 数据的操作实际上调用的都是 `Store` 类：

```
protected function buildSession($handler)
{
    if ($this->app['config']['session.encrypt']) {
        return $this->buildEncryptedSession($handler);
    } else {
        return new Store($this->app['config']['session.cookie'],
 $handler);
    }
}

protected function buildEncryptedSession($handler)
{
    return new EncryptedStore(
        $this->app['config']['session.cookie'], $handler, $this-
>app['encrypter']
    );
}

public function __call($method, $parameters)
{
    return $this->driver()->$method(...$parameters);
}
```

如果需要对 `session` 进行加密，那么就会创建一个 `EncryptedStore` 类，该类继承 `Store` 类。

setId

`session` 驱动建立之后，就要进行 `sessionId` 的设置，如果 `cookie` 中存在 `sessionId`，我们就会从中获取，否则我们就需要重新生成新的 `sessionId`

```
public function setId($id)
{
    $this->id = $this->isValidId($id) ? $id : $this->generateSes
sionId();
}

public function isValidId($id)
{
    return is_string($id) && ctype_alnum($id) && strlen($id) ===
40;
}

protected function generateSessionId()
{
    return Str::random(40);
}
```

session--start

一切准备就绪后，我们就要启动 `session`，如果当前请求存在未过期 `session`，那么就要利用 `session` 驱动将数据读取出来：

```
public function start()
{
    $this->loadSession();

    if (! $this->has('_token')) {
        $this->regenerateToken();
    }

    return $this->started = true;
}

protected function loadSession()
{
    $this->attributes = array_merge($this->attributes, $this->re
adFromHandler());
}
```

`readFromHandler` 函数就是读取 `session` 的过程：

```
protected function readFromHandler()
{
    if ($data = $this->handler->read($this->getId())) {
        $data = @unserialize($this->prepareForUnserialize($data)
);

        if ($data !== false && ! is_null($data) && is_array($dat
a)) {
            return $data;
        }
    }

    return [];
}
```

- 未加密 session 数据的加载

对于未加密的 `session` 来说，`prepareForUnserialize` 直接返回了数据：

```
protected function prepareForUnserialize($data)
{
    return $data;
}
```

- 加密 session 数据

```
protected function prepareForUnserialize($data)
{
    try {
        return $this->encrypter->decrypt($data);
    } catch (DecryptException $e) {
        return serialize([]);
    }
}
```

- file 驱动

```
public function read($sessionId)
{
    if ($this->files->exists($path = $this->path.'/'.$sessionId)
) {
        if (filemtime($path) >= Carbon::now()->subMinutes($this-
>minutes)->getTimestamp()) {
            return $this->files->get($path, true);
        }
    }

    return '';
}
```

- redis 驱动

```
public function read($sessionId)
{
    return $this->cache->get($sessionId, '');
}
```

session 垃圾回收

`session` 的垃圾回收用于随机性地删除旧 `session` 数据。由于某些驱动，例如 `FileSessionHandler`，程序不会定期删除那些已经过时的 `session` 文件，那么 `session` 文件一定会越来越多，所以我们就需要一种垃圾回收机制：

```
protected function collectGarbage(Session $session)
{
    $config = $this->manager->getSessionConfig();

    if ($this->configHitsLottery($config)) {
        $session->getHandler()->gc($this->getSessionLifetimeInSe
conds());
    }
}

protected function configHitsLottery(array $config)
{
    return random_int(1, $config['lottery'][1]) <= $config['lott
ery'][0];
}
```

`configHitsLottery` 函数就是判断当前是否被随机要进行垃圾回收任务。这种随机性概率由 `lottery` 来设置。

`FileSessionHandler` 的垃圾回收：

```
public function gc($lifetime)
{
    $files = Finder::create()
                ->in($this->path)
                ->files()
                ->ignoreDotFiles(true)
                ->date('<= now - '.$lifetime.' seconds');

    foreach ($files as $file) {
        $this->files->delete($file->getRealPath());
    }
}
```

存储前一页

很多时候我们都需要从 `session` 中获取前一页的地址，例如用户授权失败就会返回上一页等等情景。

```
protected function storeCurrentUrl(Request $request, $session)
{
    if ($request->method() === 'GET' && $request->route() && ! $
request->ajax()) {
        $session->setPreviousUrl($request->fullUrl());
    }
}

public function setPreviousUrl($url)
{
    $this->put('_previous.url', $url);
}
```

中间件的结束

当请求结束时，会调用中间件的 `terminate` 函数，这里程序会将新的 `session` 数据持久化到各个驱动器中：

```
public function terminate($request, $response)
{
    if ($this->sessionHandled && $this->sessionConfigured() && !
 $this->usingCookieSessions()) {
        $this->manager->driver()->save();
    }
}
```

`session` 的保存：

```
public function save()
{
    $this->ageFlashData();

    $this->handler->write($this->getId(), $this->prepareForStora
ge(
        serialize($this->attributes)
    ));

    $this->started = false;
}
```

`session` 的保存会删除需要 `flash` 的闪存数据，也就是只想用于下一次请求的数据：

```
public function ageFlashData()
{
    $this->forget($this->get('_flash.old', []));

    $this->put('_flash.old', $this->get('_flash.new', []));

    $this->put('_flash.new', []);
}
```

对于不加密的数据，保存前的 `prepareForStorage` 不会对数据进行任何操作：

```
protected function prepareForStorage($data)
{
    return $data;
}
```

对于加密的数据，则需要事先加密：

```
protected function prepareForStorage($data)
{
    return $this->encrypter->encrypt($data);
}
```

session 数据操作

get 函数

当我们想要获取 `session` 中的数据时，我们经常使用 `get` 方法

```
public function show(Request $request, $id)
{
    $value = $request->session()->get('key');

    //
}
```

`get` 方法首先会调用 `sessionManager` 的魔术方法：

```
public function __call($method, $parameters)
{
    return $this->driver()->$method(...$parameters);
}
```

`driver` 函数会返回 `Store` 对象，调用 `get` 方法

```
public function get($key, $default = null)
{
    return Arr::get($this->attributes, $key, $default);
}
```

我们从上一节知道，在 `startSession` 中间件启动后，`session` 数据已经加载到了 `store` 对象中，因此获取数据很简单：

```
public function get($key, $default = null)
{
    return Arr::get($this->attributes, $key, $default);
}
```

all 函数

`all` 函数可以取出所有的 `session` 数据

```
public function all()
{
    return $this->attributes;
}
```

has 函数

要确定 Session 中是否存在某个值，可以使用 `has` 方法。如果该值存在且不为 `null`，那么 `has` 方法会返回 `true`：

```
public function has($key)
{
    return ! collect(is_array($key) ? $key : func_get_args())->c
ontains(function ($key) {
        return is_null($this->get($key));
    });
}
```

exists 函数

要确定 `Session` 中是否存在某个值，即使其值为 `null` ，也可以使用 `exists` 方法。如果值存在，则 `exists` 方法返回 `true`

```
public function exists($key)
{
    return ! collect(is_array($key) ? $key : func_get_args())->c
ontains(function ($key) {
        return ! Arr::exists($this->attributes, $key);
    });
}
```

put 方法

要存储数据到 `Session` ，可以使用 `put` 方法

```
public function put($key, $value = null)
{
    if (! is_array($key)) {
        $key = [$key => $value];
    }

    foreach ($key as $arrayKey => $arrayValue) {
        Arr::set($this->attributes, $arrayKey, $arrayValue);
    }
}
```

push 方法

`push` 方法可以将一个新的值添加到 `Session` 数组内。

```
public function push($key, $value)
{
    $array = $this->get($key, []);

    $array[] = $value;

    $this->put($key, $array);
}
```

remember 方法

`remember` 方法用于有即取，无即存的情况：

```
public function remember($key, Closure $callback)
{
    if (! is_null($value = $this->get($key))) {
        return $value;
    }

    return tap($callback(), function ($value) use ($key) {
        $this->put($key, $value);
    });
}
```

increment 方法

`increment` 方法用于增加某 `session` 数据的值：

```
public function increment($key, $amount = 1)
{
    $this->put($key, $value = $this->get($key, 0) + $amount);

    return $value;
}
```

decrement 方法

```
public function decrement($key, $amount = 1)
{
    return $this->increment($key, $amount * -1);
}
```

pull 方法

`pull` 方法可以只用一条语句就从 `Session` 检索并且删除一个项目：

```
public function pull($key, $default = null)
{
    return Arr::pull($this->attributes, $key, $default);
}
```

flash 闪存数据

有时候你仅想在下一个请求之前在 `Session` 中存入数据，你可以使用 `flash` 方法。使用这个方法保存在 `session` 中的数据，只会保留到下个 `HTTP` 请求到来之前，然后就会被删除。闪存数据主要用于短期的状态消息

```
public function flash($key, $value)
{
    $this->put($key, $value);

    $this->push('_flash.new', $key);

    $this->removeFromOldFlashData([$key]);
}

protected function removeFromOldFlashData(array $keys)
{
    $this->put('_flash.old', array_diff($this->get('_flash.old',
 []), $keys));
}
```

闪存数据的实现很简单，`session` 中会维护两个数组：`_flash.new`、`_flash.old`，每次 `session` 结束前，都会删除 `_flash.old` 中的存储的 `key` 对应存储在 `session` 的 `value`。

now 方法

`now` 方法用于存储只有本次请求采用的数据

```
 public function now($key, $value)
{
    $this->put($key, $value);

    $this->push('_flash.old', $key);
}
```

reflash 方法

如果需要保留闪存数据给更多请求，可以使用 reflash 方法，这将会将所有的闪存数据保留给其他请求。

```
public function reflash()
{
    $this->mergeNewFlashes($this->get('_flash.old', []));

    $this->put('_flash.old', []);
}
```

这样，`_flash.old` 中的数据就会被合并到 `_flash.new` 中。

keep 方法

只想保留特定的闪存数据给更多请求，则可以使用 keep 方法:

```
public function keep($keys = null)
{
    $this->mergeNewFlashes($keys = is_array($keys) ? $keys : fun
c_get_args());

    $this->removeFromOldFlashData($keys);
}
```

forget 方法

forget 方法可以从 Session 内删除一条数据。

```
public function forget($keys)
{
    Arr::forget($this->attributes, $keys);
}
```

flush 方法

如果你想删除 Session 内所有数据，可以使用 flush 方法：

```
public function flush()
{
    $this->attributes = [];
}
```

重新生成 Session ID

重新生成 `Session ID`，通常是为了防止恶意用户利用 `session fixation` 对应用进行攻击。如果使用了内置函数 `LoginController`，Laravel 会自动重新生成身份验证中 `Session ID`。否则，你需要手动使用 `regenerate` 方法重新生成 `Session ID`。

```
public function regenerate($destroy = false)
{
    return $this->migrate($destroy);
}

public function migrate($destroy = false)
{
    if ($destroy) {
        $this->handler->destroy($this->getId());
    }

    $this->setExists(false);

    $this->setId($this->generateSessionId());

    return true;
}
```

Laravel Event——事件系统的启动与运行源码分析

前言

`Laravel` 的事件系统是一个简单的观察者模式，主要目的是用于代码的解耦，可以防止不同功能的代码耦合在一起。`laravel` 中事件系统由两部分构成，一个是事件的名称，事件的名称可以是个字符串，例如 `event.email`，也可以是一个事件类，例如 `App\Events\OrderShipped`；另一个是事件的 `listener`，可以是一个闭包，还可以是监听类，例如 `App\Listeners\SendShipmentNotification`。

事件服务的注册

事件服务的注册分为两部分，一个是 `Application` 启动时所调用的 `registerBaseServiceProviders` 函数：

```
protected function registerBaseServiceProviders()
{
    $this->register(new EventServiceProvider($this));

    $this->register(new LogServiceProvider($this));

    $this->register(new RoutingServiceProvider($this));
}
```

其中的 `EventServiceProvider` 是 `/Illuminate/Events/EventServiceProvider`：

```
public function register()
{
    $this->app->singleton('events', function ($app) {
        return (new Dispatcher($app))->setQueueResolver(function
() use ($app) {
            return $app->make(QueueFactoryContract::class);
        });
    });
}
```

这部分为 `Ioc` 容器注册了 `events` 实例，`Dispatcher` 就是 `events` 真正的实现类。`QueueResolver` 是队列化事件的实现。

另一个注册是普通注册类 `/app/Providers/EventServiceProvider` ：

```
class EventServiceProvider extends ServiceProvider
{
    protected $listen = [
        'App\Events\SomeEvent' => [
            'App\Listeners\EventListener',
        ],
    ];

    public function boot()
    {
        parent::boot();
        //
    }
}
```

这个注册类的主要作用是事件系统的启动，这个类继承自 `/Illuminate/Foundation/Support/Providers/EventServiceProvider` :

```php
class EventServiceProvider extends ServiceProvider
{
    protected $listen = [];

    protected $subscribe = [];

    public function boot()
    {
        foreach ($this->listens() as $event => $listeners) {
            foreach ($listeners as $listener) {
                Event::listen($event, $listener);
            }
        }

        foreach ($this->subscribe as $subscriber) {
            Event::subscribe($subscriber);
        }
    }
}
```

可以看到，事件系统的启动主要是进行事件系统的监听与订阅。

事件系统的监听 listen

所谓的事件监听，就是将事件名与闭包函数，或者事件类与监听类之间建立关联。

```
public function listen($events, $listener)
{
    foreach ((array) $events as $event) {
        if (Str::contains($event, '*')) {
            $this->setupWildcardListen($event, $listener);
        } else {
            $this->listeners[$event][] = $this->makeListener($li
stener);
        }
    }
}

protected function setupWildcardListen($event, $listener)
{
    $this->wildcards[$event][] = $this->makeListener($listener,
true);
}
```

对于有通配符的事件名，会统一放入 `wildcards` 数组中，`makeListener` 是创建事件的关键:

```
public function makeListener($listener, $wildcard = false)
{
    if (is_string($listener)) {
        return $this->createClassListener($listener, $wildcard);
    }

    return function ($event, $payload) use ($listener, $wildcard)
 {
        if ($wildcard) {
            return $listener($event, $payload);
        } else {
            return $listener(...array_values($payload));
        }
    };
}
```

创建监听者的时候，会判断监听对象是监听类还是闭包函数。

对于闭包监听来说， `makeListener` 会再包上一层闭包函数，根据是否含有通配符来确定具体的参数。

对于监听类来说，会继续 `createClassListener` :

```
public function createClassListener($listener, $wildcard = false)

{
    return function ($event, $payload) use ($listener, $wildcard)
 {
        if ($wildcard) {
            return call_user_func($this->createClassCallable($li
stener), $event, $payload);
        } else {
            return call_user_func_array(
                $this->createClassCallable($listener), $payload
            );
        }
    };
}

protected function createClassCallable($listener)
{
    list($class, $method) = $this->parseClassCallable($listener)
;

    if ($this->handlerShouldBeQueued($class)) {
        return $this->createQueuedHandlerCallable($class, $metho
d);
    } else {
        return [$this->container->make($class), $method];
    }
}
```

对于监听类来说，程序首先会判断监听类对应的函数：

```
protected function parseClassCallable($listener)
{
    return Str::parseCallback($listener, 'handle');
}
```

如果未指定监听类的对应函数，那么会默认 `handle` 函数。

如果当前监听类是队列的话，会将任务推送给队列。

触发事件

事件的触发可以利用事件名，或者事件类的实例：

```
public function dispatch($event, $payload = [], $halt = false)
{
    list($event, $payload) = $this->parseEventAndPayload(
        $event, $payload
    );

    if ($this->shouldBroadcast($payload)) {
        $this->broadcastEvent($payload[0]);
    }

    $responses = [];

    foreach ($this->getListeners($event) as $listener) {
        $response = $listener($event, $payload);

        if (! is_null($response) && $halt) {
            return $response;
        }

        if ($response === false) {
            break;
        }

        $responses[] = $response;
    }

    return $halt ? null : $responses;
}
```

`parseEventAndPayload` 函数利用传入参数是事件名还是事件类实例来确定监听类函数的参数：

```
protected function parseEventAndPayload($event, $payload)
{
    if (is_object($event)) {
        list($payload, $event) = [[$event], get_class($event)];
    }

    return [$event, array_wrap($payload)];
}
```

如果是事件类的实例，那么监听函数的参数就是事件类自身；如果是事件类名，那么监听函数的参数就是触发事件时传入的参数。

获得事件与参数后，就要获取监听类：

```
public function getListeners($eventName)
{
    $listeners = isset($this->listeners[$eventName]) ? $this->listeners[$eventName] : [];

    $listeners = array_merge(
        $listeners, $this->getWildcardListeners($eventName)
    );

    return class_exists($eventName, false)
                ? $this->addInterfaceListeners($eventName, $listeners)
                : $listeners;
}
```

寻找监听类的时候，也要从通配符监听器中寻找：

```
protected function getWildcardListeners($eventName)
{
    $wildcards = [];

    foreach ($this->wildcards as $key => $listeners) {
        if (Str::is($key, $eventName)) {
            $wildcards = array_merge($wildcards, $listeners);
        }
    }

    return $wildcards;
}
```

如果监听类继承自其他类，那么父类也会一并当做监听类返回。

获得了监听类之后，就要调用监听类相应的函数。

触发事件时有一个参数 `halt` ，这个参数如果是 `true` 的时候，只要有一个监听类返回了结果，那么就会立刻返回。例如：

```
public function testHaltingEventExecution()
{
    unset($_SERVER['__event.test']);
    $d = new Dispatcher;
    $d->listen('foo', function ($foo) {
        $this->assertTrue(true);

        return 'here';
    });
    $d->listen('foo', function ($foo) {
        throw new Exception('should not be called');
    });
    $d->until('foo', ['bar']);
}
```

多个监听类在运行的时候，只要有一个返回了 `false` ，那么就会中断事件。

push 函数

`push` 函数可以将触发事件的参数事先设置好，这样触发的时候只要写入事件名即可，例如：

```
public function testQueuedEventsAreFired()
{
    unset($_SERVER['__event.test']);
    $d = new Dispatcher;
    $d->push('update', ['name' => 'taylor']);
    $d->listen('update', function ($name) {
        $_SERVER['__event.test'] = $name;
    });

    $this->assertFalse(isset($_SERVER['__event.test']));
    $d->flush('update');
    $this->assertEquals('taylor', $_SERVER['__event.test']);
}
```

原理也很简单：

```
public function push($event, $payload = [])
{
    $this->listen($event.'_pushed', function () use ($event, $pa
yload) {
        $this->dispatch($event, $payload);
    });
}

public function flush($event)
{
    $this->dispatch($event.'_pushed');
}
```

数据库 Eloquent 的事件

数据库模型的事件的注册除了以上的方法还有另外两种，具体详情可以看：Laravel 模型事件实现原理；

事件注册

- 静态方法定义

```
class EventServiceProvider extends ServiceProvider
{
    public function boot()
    {
        parent::boot();

        User::saved(function(User$user) {

        });

        User::saved('UserSavedListener@saved');
    }
}
```

- 观察者

```
class UserObserver
{
    public function created(User $user)
    {
        //
    }

    public function saved(User $user)
    {
        //
    }
}
```

然后在某个服务提供者的boot方法中注册观察者：

```
class AppServiceProvider extends ServiceProvider
{
    public function boot()
    {
        User::observe(UserObserver::class);
    }

    public function register()
    {
        //
    }
}
```

这两种方法都是向事件系统注册事件名 `eloquent.{$event}: {static::class}`：

- 静态方法

```
public static function saved($callback)
{
    static::registerModelEvent('saved', $callback);
}

protected static function registerModelEvent($event, $callback)
{
    if (isset(static::$dispatcher)) {
        $name = static::class;

        static::$dispatcher->listen("eloquent.{$event}: {$name}"
, $callback);
    }
}
```

- 观察者

```
public static function observe($class)
{
    $instance = new static;

    $className = is_string($class) ? $class : get_class($class);

    foreach ($instance->getObservableEvents() as $event) {
        if (method_exists($class, $event)) {
            static::registerModelEvent($event, $className.'@'.$e
vent);
        }
    }
}

public function getObservableEvents()
{
    return array_merge(
        [
            'creating', 'created', 'updating', 'updated',
            'deleting', 'deleted', 'saving', 'saved',
            'restoring', 'restored',
        ],
        $this->observables
    );
}
```

事件触发

模型事件的触发需要调用 `fireModelEvent` 函数：

```
protected function fireModelEvent($event, $halt = true)
{
    if (! isset(static::$dispatcher)) {
        return true;
    }

    $method = $halt ? 'until' : 'fire';

    $result = $this->fireCustomModelEvent($event, $method);

    return ! is_null($result) ? $result : static::$dispatcher->{
$method}(
        "eloquent.{$event}: ".static::class, $this
    );
}
```

`fireCustomModelEvent` 是我们本文中着重讲的事件类与监听类的触发：

```
protected function fireCustomModelEvent($event, $method)
{
    if (! isset($this->events[$event])) {
        return;
    }

    $result = static::$dispatcher->$method(new $this->events[$ev
ent]($this));

    if (! is_null($result)) {
        return $result;
    }
}
```

如果没有对应的事件后，会继续利用事件名进行触发。

`until` 是我们上一节讲的如果任意事件返回正确结果，就会直接返回，不会继续进行下一个事件。

Laravel Queue——消息队列任务与分发源码剖析

前言

在实际的项目开发中，我们经常会遇到需要轻量级队列的情形，例如发短信、发邮件等，这些任务不足以使用 `kafka` 、 `RabbitMQ` 等重量级的消息队列，但是又的确需要异步、重试、并发控制等功能。通常来说，我们经常会使用 `Redis` 、 `Beanstalk` 、 `Amazon SQS` 来实现相关功能， `laravel` 为此对不同的后台队列服务提供统一的 `API` ，本文将会介绍应用最为广泛的 `redis` 队列。

本文参考文档资料：

使用 Laravel Queue 不得不明白的知识

Laravel 的消息队列剖析

背景知识

在讲解 `laravel` 的队列服务之前，我们要先说说基于 `redis` 的队列服务。首先，redis设计用来做缓存的，但是由于它自身的某种特性使得它可以用来做消息队列，

redis 队列的数据结构

- List 链表

`redis` 做消息队列的特性例如FIFO（先入先出）很容易实现，只需要一个 `list` 对象从头取数据，从尾部塞数据即可。

相关的命令：（1）左侧入右侧出：lpush/rpop；（2）右侧入左侧出：rpush/lpop。

这个简单的消息队列很容易实现。

- Zset 有序集合

有些任务场景，并不需要任务立刻执行，而是需要延迟执行；有些任务很重要，需要在任务失败的时候重新尝试。这些功能仅仅依靠 `list` 是无法完成的。这个时候，就需要 `redis` 的有序集合。

`Redis` 有序集合和 `Redis` 集合类似，是不包含相同字符串的合集。它们的差别是，每个有序集合的成员都关联着一个评分 `score` ，这个评分用于把有序集合中的成员按最低分到最高分排列。

单看有序集合和延迟任务并无关系，但是可以将有序集合的评分 `score` 设置为延时任务开启的时间，之后轮询这个有序集合，将到期的任务拿出来进行处理，这样就实现了延迟任务的功能。

对于重要的需要重试的任务，在任务执行之前，会将该任务放入有序集合中，设置任务最长的执行时间。若任务顺利执行完毕，该任务会在有序集合中删除。如果任务没有在规定时间内完成，那么该有序集合的任务将会被重新放入队列中。

相关命令：

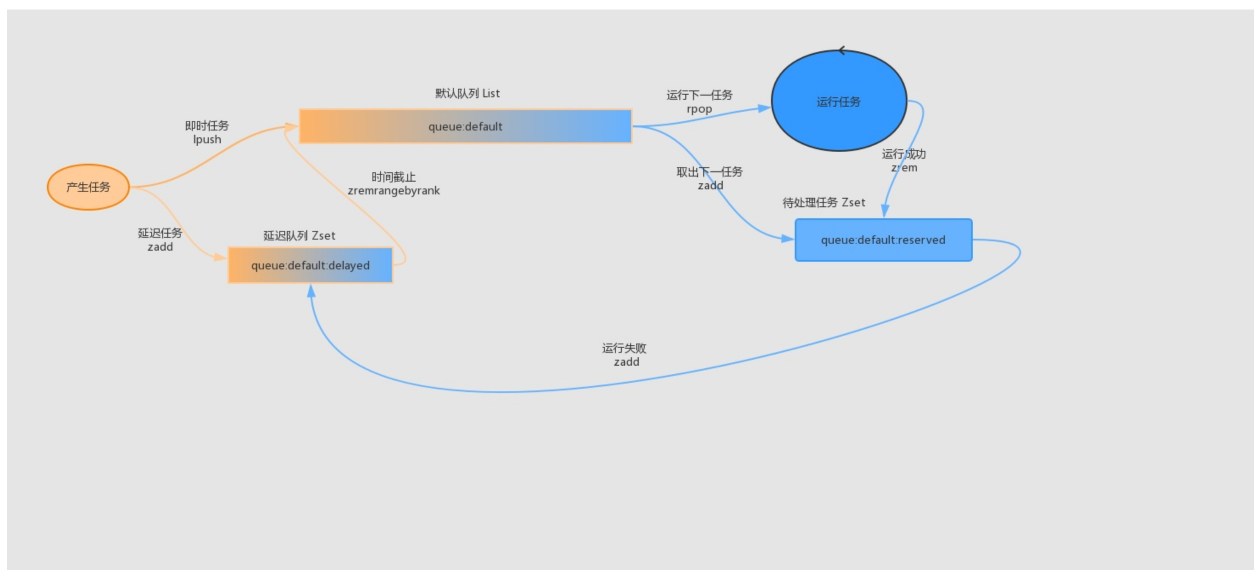
(1) `ZADD` 添加一个或多个成员到有序集合，或者如果它已经存在更新其分数。

(2) `ZRANGEBYSCORE` 按分数返回一个成员范围的有序集合。

(3) `ZREMRANGEBYRANK` 在给定的索引之内删除所有成员的有序集合。

laravel 队列服务的任务调度

队列服务的任务调度过程如下：



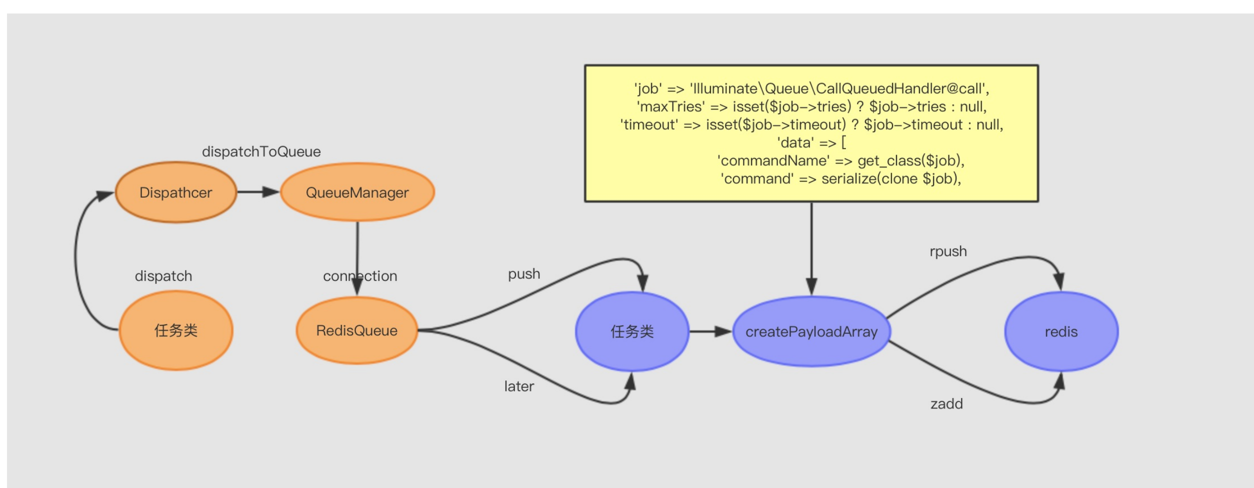
`laravel` 的队列服务由两个进程控制，一个是生产者，一个是消费者。这两个进程操纵了 `redis` 三个队列，其中一个 `List` ，负责即时任务，两个 `Zset` ，负责延时任务与待处理任务。

生产者负责向 `redis` 推送任务，如果是即时任务，默认就会向 `queue:default` 推送；如果是延时任务，就会向 `queue:default:delayed` 推送。

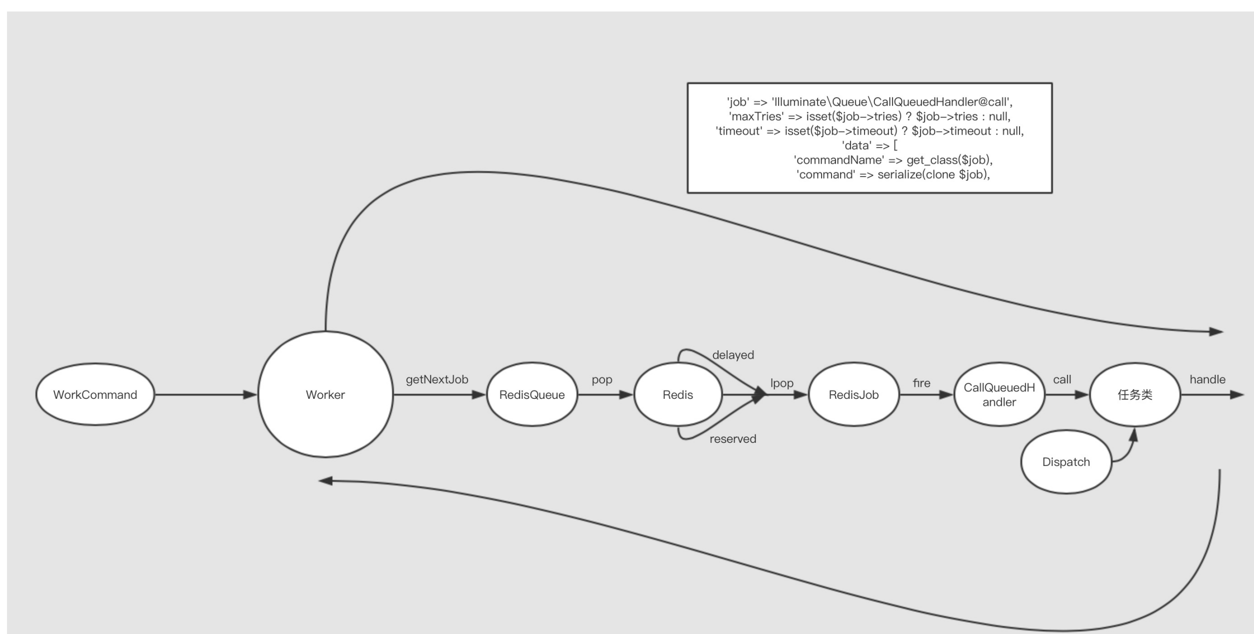
消费者轮询两个队列，不断的从队列中取出任务，先把任务放入 `queue:default:reserved` 中，再执行相关任务。如果任务执行成功，就会删除 `queue:default:reserved` 中的任务，否则会被重新放入 `queue:default:delayed` 队列中。

laravel 队列服务的总体流程

任务分发流程：



任务处理器运作：



laravel 队列服务的注册与启动

`laravel` 队列服务需要注册的服务比较多：

```
class QueueServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->registerManager();

        $this->registerConnection();

        $this->registerWorker();

        $this->registerListener();

        $this->registerFailedJobServices();
    }
}
```

registerManager 注册门面

`registerManager` 负责注册队列服务的门面类：

```
protected function registerManager()
{
    $this->app->singleton('queue', function ($app) {
        return tap(new QueueManager($app), function ($manager) {
            $this->registerConnectors($manager);
        });
    });
}

public function registerConnectors($manager)
{
    foreach (['Null', 'Sync', 'Database', 'Redis', 'Beanstalkd',
'Sqs'] as $connector) {
        $this->{"register{$connector}Connector"}($manager);
    }
}

protected function registerRedisConnector($manager)
{
    $manager->addConnector('redis', function () {
        return new RedisConnector($this->app['redis']);
    });
}
```

`QueueManager` 是队列服务的总门面，提供一切与队列相关的操作接口。`QueueManager` 中有一个成员变量 `$connectors` ，该成员变量中存储着所有 `laravel` 支持的底层队列服务：'Database', 'Redis', 'Beanstalkd', 'Sqs'。

```
class QueueManager implements FactoryContract, MonitorContract
{
    public function addConnector($driver, Closure $resolver)
    {
        $this->connectors[$driver] = $resolver;
    }
}
```

成员变量 `$connectors` 会被存储各种驱动的 `connector`，例如 `RedisConnector`、`SqsConnector`、`DatabaseConnector`、`BeanstalkdConnector`。

registerConnection 底层队列连接服务

接下来，就要连接实现队列的底层服务了，例如 `redis`：

```
protected function registerConnection()
{
    $this->app->singleton('queue.connection', function ($app) {
        return $app['queue']->connection();
    });
}

public function connection($name = null)
{
    $name = $name ?: $this->getDefaultDriver();

    if (! isset($this->connections[$name])) {
        $this->connections[$name] = $this->resolve($name);

        $this->connections[$name]->setContainer($this->app);
    }

    return $this->connections[$name];
}

public function getDefaultDriver()
{
    return $this->app['config']['queue.default'];
}
```

`connection` 函数首先会获取 `连接` 名，没有 `连接` 名就会从 `config` 中获取默认的连接。

```
protected function resolve($name)
{
    $config = $this->getConfig($name);

    return $this->getConnector($config['driver'])
                ->connect($config)
                ->setConnectionName($name);
}
```

`resolve` 函数利用相应的底层驱动 `connector` 进行连接操作，也就是 `connect` 函数，该函数会返回 `RedisQueue` ：

```
class RedisConnector implements ConnectorInterface
{
    public function connect(array $config)
    {
        return new RedisQueue(
            $this->redis, $config['queue'],
            Arr::get($config, 'connection', $this->connection),
            Arr::get($config, 'retry_after', 60)
        );
    }
}
```

registerWorker 消费者服务注册

消费者的注册服务会返回 `Illuminate\Queue\Worker` 类：

```
protected function registerWorker()
{
    $this->app->singleton('queue.worker', function () {
        return new Worker(
            $this->app['queue'], $this->app['events'], $this->ap
p[ExceptionHandler::class]
        );
    });
}
```

laravel Bus 服务注册与启动

定义好自己想要的队列类之后，还需要将队列任务推送给底层驱动后台，例如 `redis` ，一般会使用 `dispatch` 函数：

```
Job::dispatch();
```

或者

```
$job = (new ProcessPodcast($pocast));

dispatch($job);
```

`dispatch` 函数就是 `Bus` 服务，专门用于分发队列任务。

```
class BusServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->app->singleton(Dispatcher::class, function ($app)
{
            return new Dispatcher($app, function ($connection =
null) use ($app) {
                return $app[QueueFactoryContract::class]->connec
tion($connection);
            });
        });

        $this->app->alias(
            Dispatcher::class, DispatcherContract::class
        );

        $this->app->alias(
            Dispatcher::class, QueueingDispatcherContract::class
        );
    }
}
```

创建任务

queue 设置

```
'redis' => [
        'driver' => 'redis',
        'connection' => 'default',
        'queue' => 'default',
        'retry_after' => 90,
    ],
```

一般来说，默认的 `redis` 配置如上， `connection` 是 `database` 中 `redis` 的连接名称； `queue` 是 `redis` 中的队列名称，值得注意的是，如果使用的是 `redis` 集群的话，这个需要使用 `key hash tag` ，也就是 `{default}` ;当任务运行超过 `retry_after` 这个时间后，该任务会被重新放入队列当中。

任务类的创建

- 任务类的结构很简单，一般来说只会包含一个让队列用来调用此任务的 `handle` 方法。
- 如果想要使得任务被推送到队列中，而不是同步执行，那么需要实现 `Illuminate\Contracts\Queue\ShouldQueue` 接口。
- 如果想要让任务推送到特定的连接中，例如 `redis` 或者 `sqs` ，那么需要设置 `conneciton` 变量。
- 如果想要让任务推送到特定的队列中去，可以设置 `queue` 变量。
- 如果想要让任务延迟推送，那么需要设置 `delay` 变量。
- 如果想要设置任务至多重试的次数，可以使用 `tries` 变量；
- 如果想要设置任务可以运行的最大秒数，那么可以使用 `timeout` 参数。
- 如果想要手动访问队列，可以使用 `trait` : `Illuminate\Queue\InteractsWithQueue` 。

- 如果队列监听器任务执行次数超过在工作队列中定义的最大尝试次数，监听器的 `failed` 方法将会被自动调用。 `failed` 方法接受事件实例和失败的异常作为参数：

```
class ProcessPodcast implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesM
odels;

    protected $podcast;

    public $connection = 'redis';

    public $queue = 'test';

    public $delay = 30;

    public $tries = 5;

    public $timeout = 30;

    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }

    public function handle(AudioProcessor $processor)
    {
        // Process uploaded podcast...

        if (false) {
            $this->release(30);
        }
    }

    public function failed(OrderShipped $event, $exception)
    {
        //
    }
}
```

任务事件

```
class AppServiceProvider extends ServiceProvider
{
    public function boot()
    {
        //任务运行前
        Queue::before(function (JobProcessing $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });

        //任务运行后
        Queue::after(function (JobProcessed $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });

        //任务循环前
        Queue::looping(function () {
              while (DB::transactionLevel() > 0) {
              DB::rollBack();
            }
        });

        //任务失败后
         Queue::failing(function (JobFailed $event) {
            // $event->connectionName
            // $event->job
            // $event->exception
        });

        //异常发生后
        Queue::exceptionOccurred(function (JobFailed $event) {
            // $event->connectionName
            // $event->job
            // $event->exception
        });
    }
```

```
}
```

任务的分发

分发服务

- 写好任务类后，就能通过 `dispatch` 辅助函数来分发它了。唯一需要传递给 `dispatch` 的参数是这个任务类的实例：

```
class PodcastController extends Controller
{
    public function store(Request $request)
    {
        // 创建播客...

        ProcessPodcast::dispatch($podcast);
    }
}
```

- 如果想延迟执行一个队列中的任务，可以用任务实例的 `delay` 方法。

```
 ProcessPodcast::dispatch($podcast)
               ->delay(Carbon::now()->addMinutes(10));
```

- 通过推送任务到不同的队列，可以给队列任务分类，甚至可以控制给不同的队列分配多少任务。要指定队列的话，就调用任务实例的 `onQueue` 方法：

```
ProcessPodcast::dispatch($podcast)->onQueue('processing');
```

- 如果使用了多个队列连接，可以将任务推到指定连接。要指定连接的话，可以在分发任务的时候使用 `onConnection` 方法：

```
ProcessPodcast::dispatch($podcast)->onConnection('redis
');
```

这些链式的函数是在 `trait` : `Illuminate\Foundation\Bus\Dispatchable` 的基础上应用的，该 `trait` 由 `dispatch` 函数启动：

```
trait Dispatchable
{
    public static function dispatch()
    {
        return new PendingDispatch(new static(...func_get_args()
));
    }
}
```

`PendingDispatch` 类中定义了链式函数，该函数巧妙在析构函数中，析构函数自动调用全局函数 `dispatch` ：

```
class PendingDispatch
{
    public function __construct($job)
    {
        $this->job = $job;
    }


    public function onConnection($connection)
    {
        $this->job->onConnection($connection);

        return $this;
    }

    public function onQueue($queue)
    {
        $this->job->onQueue($queue);

        return $this;
    }

    public function delay($delay)
    {
        $this->job->delay($delay);

        return $this;
    }

    public function __destruct()
    {
        dispatch($this->job);
    }


}
```

各个函数里面的 `onConnection` 、 `delay` 、 `onQueue` 等函数是任务中的 `trait` ： `Illuminate\Bus\Queueable`

```
trait Queueable
{
    public function onConnection($connection)
    {
        $this->connection = $connection;

        return $this;
    }

    public function onQueue($queue)
    {
        $this->queue = $queue;

        return $this;
    }

    public function delay($delay)
    {
        $this->delay = $delay;

        return $this;
    }
}
```

dispatch 任务分发源码

任务的分发离不开 `Bus` 服务，可以利用全局函数 `dispatch` ，还可以使用 `Dispatchable` 这个 `trait` :

```
class Dispatcher implements QueueingDispatcher
{
     public function dispatch($command)
    {
        if ($this->queueResolver && $this->commandShouldBeQueued
($command)) {
            return $this->dispatchToQueue($command);
        } else {
            return $this->dispatchNow($command);
        }
    }

    protected function commandShouldBeQueued($command)
    {
        return $command instanceof ShouldQueue;
    }
}
```

我们这里主要看异步的任务：

```
public function dispatchToQueue($command)
{
    $connection = isset($command->connection) ? $command->connec
tion : null;

    $queue = call_user_func($this->queueResolver, $connection);

    if (! $queue instanceof Queue) {
        throw new RuntimeException('Queue resolver did not retur
n a Queue implementation.');
    }

    if (method_exists($command, 'queue')) {
        return $command->queue($queue, $command);
    } else {
        return $this->pushCommandToQueue($queue, $command);
    }
}
```

进行任务分发之前，首先要利用 `queueResolver` 连接底层驱动。如果任务类中含有 `queue` 函数，那么就会利用用户自己的 `queue` 对驱动进行推送任务。否则就会启动默认的程序：

```
protected function pushCommandToQueue($queue, $command)
{
    if (isset($command->queue, $command->delay)) {
        return $queue->laterOn($command->queue, $command->delay,
 $command);
    }

    if (isset($command->queue)) {
        return $queue->pushOn($command->queue, $command);
    }

    if (isset($command->delay)) {
        return $queue->later($command->delay, $command);
    }

    return $queue->push($command);
}
```

我们以 `redis` 为例，`queue` 这个类就是 `Illuminate\Queue\RedisQueue`：

```
class RedisQueue extends Queue implements QueueContract
{
    public function push($job, $data = '', $queue = null)
    {
        return $this->pushRaw($this->createPayload($job, $data),
 $queue);
    }

    public function pushOn($queue, $job, $data = '')
    {
        return $this->push($job, $data, $queue);
    }

    public function later($delay, $job, $data = '', $queue = nul
l)
    {
        return $this->laterRaw($delay, $this->createPayload($job
, $data), $queue);
    }

    public function laterOn($queue, $delay, $job, $data = '')
    {
        return $this->later($delay, $job, $data, $queue);
    }
}
```

我们先看 `push`，`push` 函数调用 `pushRaw`，在调用之前，要把任务类进行序列化，并且以特定的格式进行 `json` 序列化：

```
protected function createPayload($job, $data = '', $queue = null)

{
    $payload = json_encode($this->createPayloadArray($job, $data
, $queue));

    if (JSON_ERROR_NONE !== json_last_error()) {
        throw new InvalidPayloadException;
    }
```

```
    return $payload;
}

protected function createPayloadArray($job, $data = '', $queue =
 null)
{
    return is_object($job)
                ? $this->createObjectPayload($job)
                : $this->createStringPayload($job, $data);
}

protected function createObjectPayload($job)
{
    return [
        'job' => 'Illuminate\Queue\CallQueuedHandler@call',
        'maxTries' => isset($job->tries) ? $job->tries : null,
        'timeout' => isset($job->timeout) ? $job->timeout : null
,
        'data' => [
            'commandName' => get_class($job),
            'command' => serialize(clone $job),
        ],
    ];
}

protected function createStringPayload($job, $data)
{
    return ['job' => $job, 'data' => $data];
}
```

格式化数据之后，就会将 `json` 推送到 `redis` 队列中，对于非延时的任务，直接调用 `rpush` 即可：

```
public function pushRaw($payload, $queue = null, array $options
= [])
{
    $this->getConnection()->rpush($this->getQueue($queue), $payl
oad);

    return Arr::get(json_decode($payload, true), 'id');
}
```

对于延时的任务，会调用 `laterRaw` ，调用 `redis` 的有序集合 `zadd` 函数:

```
protected function availableAt($delay = 0)
{
    return $delay instanceof DateTimeInterface
                        ? $delay->getTimestamp()
                        : Carbon::now()->addSeconds($delay)->get
Timestamp();
}

protected function laterRaw($delay, $payload, $queue = null)
{
    $this->getConnection()->zadd(
        $this->getQueue($queue).':delayed', $this->availableAt($
delay), $payload
    );

    return Arr::get(json_decode($payload, true), 'id');
}
```

这样，相关任务就会被分发到 `redis` 对应的队列中去。

Laravel Queue——消息队列任务处理器源码剖析

运行队列处理器

队列处理器的设置

`Laravel` 包含一个队列处理器，当新任务被推到队列中时它能处理这些任务。你可以通过 `queue:work` 命令来运行处理器。要注意，一旦 `queue:work` 命令开始，它将一直运行，直到你手动停止或者你关闭控制台：

```
php artisan queue:work
```

- 可以指定队列处理器所使用的连接。

```
php artisan queue:work redis
```

- 可以自定义队列处理器，方式是处理给定连接的特定队列。

```
php artisan queue:work redis --queue=emails
```

- 可以使用 `--once` 选项来指定仅对队列中的单一任务进行处理：

```
php artisan queue:work --once
```

- 如果一个任务失败了，会被放入延时队列中取，`--delay` 选项可以设置失败任务的延时时间：

```
php artisan queue:work --delay=2
```

- 如果想要限制一个任务的内存，可以使用 `--memory`：

```
php artisan queue:work --memory=128
```

- 当队列需要处理任务时，进程将继续处理任务，它们之间没有延迟。但是，如果没有新的工作可用， `--sleep` 参数决定了工作进程将 「睡眠」 多长时间：

```
php artisan queue:work --sleep=3
```

- 可以指定 `Laravel` 队列处理器最多执行多长时间后就应该被关闭掉：

```
php artisan queue:work --timeout=60
```

- 可以指定 `Laravel` 队列处理器失败任务重试的次数：

```
php artisan queue:work --tries=60
```

可以看出来，队列处理器的设置大多数都可以由任务类进行设置，但是其中三个 `sleep` 、 `delay` 、 `memory` 只能由 `artisan` 来设置。

WorkCommand 命令行启动

任务处理器进程的命令行模式会调用 `Illuminate\Queue\Console\WorkCommand` ，这个类在初始化的时候依赖注入了 `Illuminate\Queue\Worker` :

```
class WorkCommand extends Command
{
    protected $signature = 'queue:work
                            {connection? : The name of connectio
n}
                            {--queue= : The queue to listen on}
                            {--daemon : Run the worker in daemon
 mode (Deprecated)}
                            {--once : Only process the next job
on the queue}
```

```
                            {--delay=0 : Amount of time to delay
 failed jobs}
                            {--force : Force the worker to run e
ven in maintenance mode}
                            {--memory=128 : The memory limit in
megabytes}
                            {--sleep=3 : Number of seconds to sl
eep when no job is available}
                            {--timeout=60 : The number of second
s a child process can run}
                            {--tries=0 : Number of times to atte
mpt a job before logging it failed}';

    public function __construct(Worker $worker)
    {
        parent::__construct();

        $this->worker = $worker;
    }

    public function fire()
    {
        if ($this->downForMaintenance() && $this->option('once')
) {
            return $this->worker->sleep($this->option('sleep'));
        }

        $this->listenForEvents();

        $connection = $this->argument('connection')
                        ?: $this->laravel['config']['queue.defau
lt'];

        $queue = $this->getQueue($connection);

        $this->runWorker(
            $connection, $queue
        );
    }
}
```

任务处理器启动后，会运行 `fire` 函数，在执行任务之前，程序首先会注册监听事件，主要监听任务完成与任务失败的情况：

```
protected function listenForEvents()
{
    $this->laravel['events']->listen(JobProcessed::class, functi
on ($event) {
        $this->writeOutput($event->job, false);
    });

    $this->laravel['events']->listen(JobFailed::class, function
($event) {
        $this->writeOutput($event->job, true);

        $this->logFailedJob($event);
    });
}

protected function writeOutput(Job $job, $failed)
{
    if ($failed) {
        $this->output->writeln('<error>['.Carbon::now()->format(
'Y-m-d H:i:s').'] Failed:</error> '.$job->resolveName());
    } else {
        $this->output->writeln('<info>['.Carbon::now()->format('
Y-m-d H:i:s').'] Processed:</info> '.$job->resolveName());
    }
}

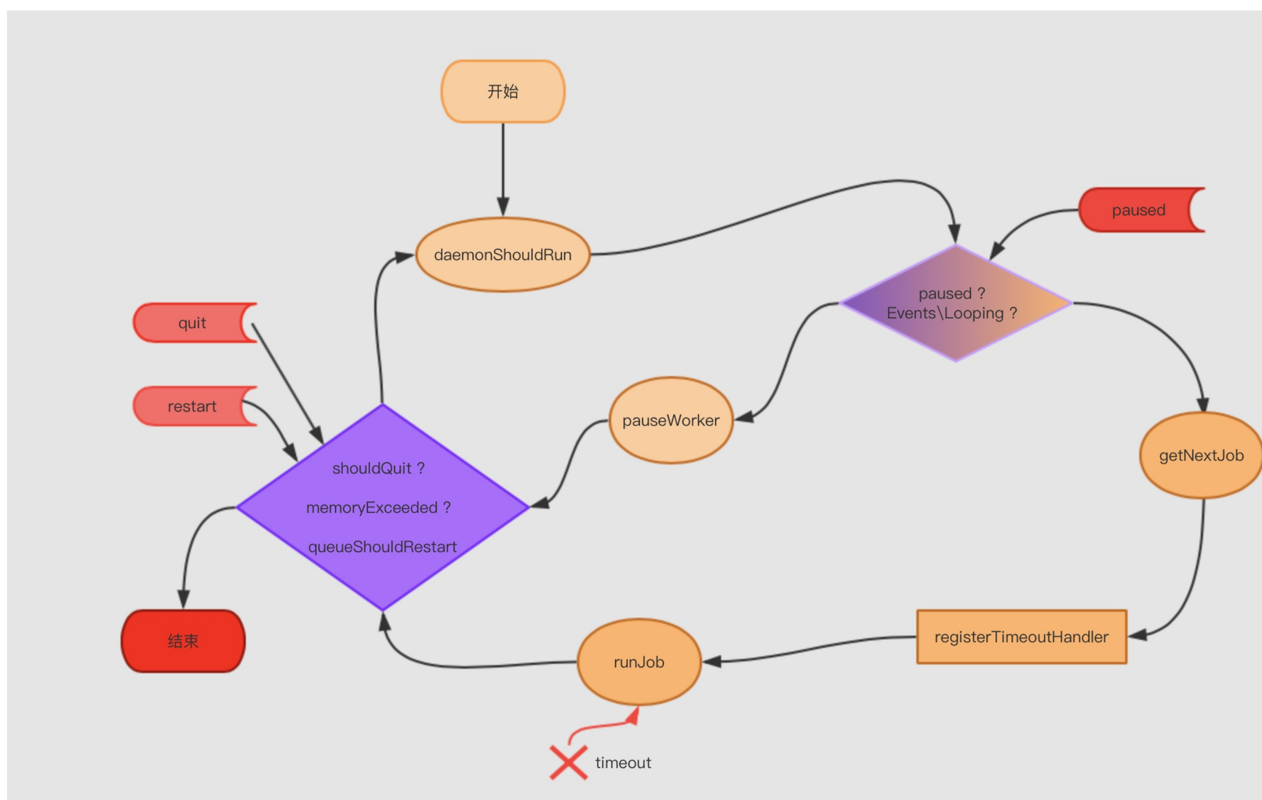
protected function logFailedJob(JobFailed $event)
{
    $this->laravel['queue.failer']->log(
        $event->connectionName, $event->job->getQueue(),
        $event->job->getRawBody(), $event->exception
    );
}
```

启动任务管理器 `runWorker`,该函数默认会调用 `Illuminate\Queue\Worker` 的 `daemon` 函数，只有在命令中强制 `--once` 参数的时候，才会执行 `runNestJob` 函数:

```
protected function runWorker($connection, $queue)
{
    $this->worker->setCache($this->laravel['cache']->driver());

    return $this->worker->{$this->option('once') ? 'runNextJob'
: 'daemon'}(
        $connection, $queue, $this->gatherWorkerOptions()
    );
}
```

Worker 任务调度



我们接下来接着看 `daemon` 函数：

```php
public function daemon($connectionName, $queue, WorkerOptions $o
ptions)
{
    $this->listenForSignals();

    $lastRestart = $this->getTimestampOfLastQueueRestart();

    while (true) {
        if (! $this->daemonShouldRun($options)) {
            $this->pauseWorker($options, $lastRestart);

            continue;
        }

        $job = $this->getNextJob(
            $this->manager->connection($connectionName), $queue
        );

        $this->registerTimeoutHandler($job, $options);

        if ($job) {
            $this->runJob($job, $connectionName, $options);
        } else {
            $this->sleep($options->sleep);
        }

        $this->stopIfNecessary($options, $lastRestart);
    }
}
```

信号处理

`listenForSignals` 函数用于 PHP 7.1 版本以上，用于脚本的信号处理。所谓的信号处理，就是由 `Process Monitor` （如 `Supervisor` ）发送并与我们的脚本进行通信的异步通知。

```
protected function listenForSignals()
{
    if ($this->supportsAsyncSignals()) {
        pcntl_async_signals(true);

        pcntl_signal(SIGTERM, function () {
            $this->shouldQuit = true;
        });

        pcntl_signal(SIGUSR2, function () {
            $this->paused = true;
        });

        pcntl_signal(SIGCONT, function () {
            $this->paused = false;
        });
    }
}

protected function supportsAsyncSignals()
{
    return version_compare(PHP_VERSION, '7.1.0') >= 0 &&
           extension_loaded('pcntl');
}
```

`pcntl_async_signals()` 被调用来启用信号处理，然后我们为多个信号注册处理程序：

- 当脚本被 `Supervisor` 指示关闭时，会引发信号 `SIGTERM`
- `SIGUSR2` 是用户定义的信号，Laravel用来表示脚本应该暂停。
- 当暂停的脚本被 `Supervisor` 指示继续进行时，会引发 `SIGCONT`

在真正运行任务之前，程序还从 `cache` 中取了一次最后一次重启的时间：

```
protected function getTimestampOfLastQueueRestart()
{
    if ($this->cache) {
        return $this->cache->get('illuminate:queue:restart');
    }
}
```

确定 `worker` 是否应该处理作业

进入循环后，首先要判断当前脚本是应该处理任务，还是应该暂停，还是应该退出：

```
protected function daemonShouldRun(WorkerOptions $options)
{
    return ! (($this->manager->isDownForMaintenance() && ! $options->force) ||
        $this->paused ||
        $this->events->until(new Events\Looping) === false);
}
```

以下几种情况，循环将不会处理任务：

- 脚本处于 `维护模式` 并且没有 `--force` 选项
- 脚本被 `supervisor` 暂停
- 脚本的 `looping` 事件监听器返回 `false`

`looping` 事件监听器在每次循环的时候都会被启动，如果返回 `false`，那么当前的循环将会被暂停：`pauseWorker`：

```
protected function pauseWorker(WorkerOptions $options, $lastRestart)
{
    $this->sleep($options->sleep > 0 ? $options->sleep : 1);

    $this->stopIfNecessary($options, $lastRestart);
}
```

脚本在 `sleep` 一段时间之后，就要重新判断当前脚本是否需要 `stop` ：

```
protected function stopIfNecessary(WorkerOptions $options, $last
Restart)
{
    if ($this->shouldQuit) {
        $this->kill();
    }

    if ($this->memoryExceeded($options->memory)) {
        $this->stop(12);
    } elseif ($this->queueShouldRestart($lastRestart)) {
        $this->stop();
    }
}

protected function queueShouldRestart($lastRestart)
{
    return $this->getTimestampOfLastQueueRestart() != $lastResta
rt;
}

protected function getTimestampOfLastQueueRestart()
{
    if ($this->cache) {
        return $this->cache->get('illuminate:queue:restart');
    }
}
```

以下情况脚本将会被 `stop` ：

- 脚本被 `supervisor` 退出
- 内存超限
- 脚本被重启过

```
public function kill($status = 0)
{
    if (extension_loaded('posix')) {
        posix_kill(getmypid(), SIGKILL);
    }

    exit($status);
}

public function stop($status = 0)
{
    $this->events->fire(new Events\WorkerStopping);

    exit($status);
}
```

脚本被重启，当前的进程需要退出并且重新加载。

获取下一个任务

当含有多个队列的时候，命令行可以用 `,` 连接多个队列的名字，位于前面的队列优先级更高：

```
protected function getNextJob($connection, $queue)
{
    try {
        foreach (explode(',', $queue) as $queue) {
            if (! is_null($job = $connection->pop($queue))) {
                return $job;
            }
        }
    } catch (Exception $e) {
        $this->exceptions->report($e);
    } catch (Throwable $e) {
        $this->exceptions->report(new FatalThrowableError($e));
    }
}
```

`$connection` 是具体的驱动，我们这里是 `Illuminate\Queue\RedisQueue`：

```
class RedisQueue extends Queue implements QueueContract
{
    public function pop($queue = null)
    {
        $this->migrate($prefixed = $this->getQueue($queue));

        list($job, $reserved) = $this->retrieveNextJob($prefixed
);

        if ($reserved) {
            return new RedisJob(
                $this->container, $this, $job,
                $reserved, $this->connectionName, $queue ?: $thi
s->default
            );
        }
    }
}

protected function getQueue($queue)
{
    return 'queues:'.($queue ?: $this->default);
}
```

在从队列中取出任务之前，需要先将 `delay` 队列和 `reserved` 队列中已经到时间的任务放到主队列中：

```
protected function migrate($queue)
{
    $this->migrateExpiredJobs($queue.':delayed', $queue);

    if (! is_null($this->retryAfter)) {
        $this->migrateExpiredJobs($queue.':reserved', $queue);
    }
}

public function migrateExpiredJobs($from, $to)
{
    return $this->getConnection()->eval(
        LuaScripts::migrateExpiredJobs(), 2, $from, $to, $this->
currentTime()
    );
}
```

由于从队列取出任务、在队列删除任务、压入主队列是三个操作，为了防止并发，程序这里使用了 `LUA` 脚本，保证三个操作的原子性：

```
public static function migrateExpiredJobs()
{
    return <<<'LUA'
          -- Get all of the jobs with an expired "score"...
        local val = redis.call('zrangebyscore', KEYS[1], '-inf',
 ARGV[1])

        -- If we have values in the array, we will remove them f
rom the first queue
        -- and add them onto the destination queue in chunks of
100, which moves
        -- all of the appropriate jobs onto the destination queu
e very safely.
        if(next(val) ~= nil) then
          redis.call('zremrangebyrank', KEYS[1], 0, #val - 1)

          for i = 1, #val, 100 do
            redis.call('rpush', KEYS[2], unpack(val, i, math.min
(i+99, #val)))
          end
        end

      return val
    LUA;
}
```

接下来，就要从主队列中获取下一个任务，在取出下一个任务之后，还要将任务放入 `reserved` 队列中，当任务执行失败后，该任务会进行重试。

```
protected function retrieveNextJob($queue)
{
    return $this->getConnection()->eval(
        LuaScripts::pop(), 2, $queue, $queue.':reserved',
        $this->availableAt($this->retryAfter)
    );
}


public static function pop()
{
    return <<<'LUA'
        -- Pop the first job off of the queue...
        local job = redis.call('lpop', KEYS[1])
        local reserved = false

        if(job ~= false) then
            -- Increment the attempt count and place job on the
reserved queue...
            reserved = cjson.decode(job)
            reserved['attempts'] = reserved['attempts'] + 1
            reserved = cjson.encode(reserved)
            redis.call('zadd', KEYS[2], ARGV[1], reserved)
        end

        return {job, reserved}
    LUA;
}
```

从 `redis` 中获取到 `job` 之后，就会将其包装成 `RedisJob` 类：

```
public function __construct(Container $container, RedisQueue $re
dis, $job, $reserved, $connectionName, $queue)
{
    $this->job = $job;
    $this->redis = $redis;
    $this->queue = $queue;
    $this->reserved = $reserved;
    $this->container = $container;
    $this->connectionName = $connectionName;

    $this->decoded = $this->payload();
}

public function payload()
{
    return json_decode($this->getRawBody(), true);
}

public function getRawBody()
{
    return $this->job;
}
```

超时处理

如果一个脚本超时， `pcntl_alarm` 将会启动并杀死当前的 `work` 进程。杀死进程后， `work` 进程将会被守护进程重启，继续进行下一个任务。

```
protected function registerTimeoutHandler($job, WorkerOptions $o
ptions)
{
    if ($options->timeout > 0 && $this->supportsAsyncSignals())
{
        pcntl_signal(SIGALRM, function () {
            $this->kill(1);
        });

        pcntl_alarm($this->timeoutForJob($job, $options) + $opti
ons->sleep);
    }
}

protected function timeoutForJob($job, WorkerOptions $options)
{
    return $job && ! is_null($job->timeout()) ? $job->timeout()
: $options->timeout;
}
```

任务事务

运行任务前后会启动两个事件 `JobProcessing` 与 `JobProcessed`，这两个事件需要事先注册监听者

```
protected function runJob($job, $connectionName, WorkerOptions $
options)
{
    try {
        return $this->process($connectionName, $job, $options);
    } catch (Exception $e) {
        $this->exceptions->report($e);
    } catch (Throwable $e) {
        $this->exceptions->report(new FatalThrowableError($e));
    }
}

public function process($connectionName, $job, WorkerOptions $op
tions)
{
    try {
        $this->raiseBeforeJobEvent($connectionName, $job);

        $this->markJobAsFailedIfAlreadyExceedsMaxAttempts(
            $connectionName, $job, (int) $options->maxTries
        );

        $job->fire();

        $this->raiseAfterJobEvent($connectionName, $job);
    } catch (Exception $e) {
        $this->handleJobException($connectionName, $job, $option
s, $e);
    } catch (Throwable $e) {
        $this->handleJobException(
            $connectionName, $job, $options, new FatalThrowableE
rror($e)
        );
    }
}
```

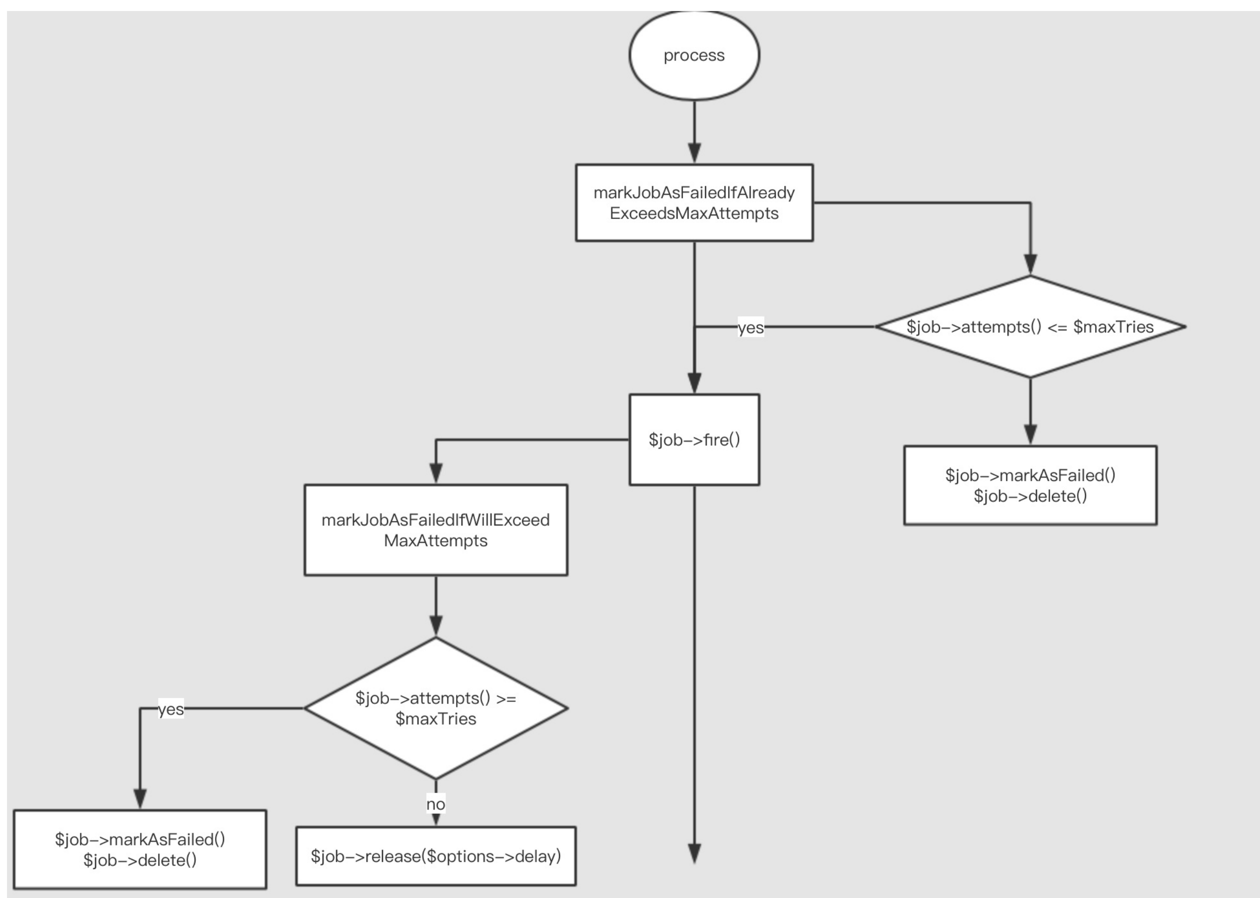
任务前与任务后事件

`raiseBeforeJobEvent` 函数用于触发任务处理前的事件，`raiseAfterJobEvent` 函数用于触发任务处理后的事件：

```
protected function raiseBeforeJobEvent($connectionName, $job)
{
    $this->events->fire(new Events\JobProcessing(
        $connectionName, $job
    ));
}

protected function raiseAfterJobEvent($connectionName, $job)
{
    $this->events->fire(new Events\JobProcessed(
        $connectionName, $job
    ));
}
```

任务异常处理



任务在运行过程中会遇到异常情况，这个时候就要判断当前任务的失败次数是不是超过限制。如果没有超过限制，那么就会把当前任务重新放回队列当中；如果超过了限制，那么就要标记当前任务为失败任务，并且将任务从 `reserved` 队列中删除。

任务失败

`markJobAsFailedIfAlreadyExceedsMaxAttempts` 函数用于任务运行前，判断当前任务是否重试次数超过限制:

```
protected function markJobAsFailedIfAlreadyExceedsMaxAttempts($c
onnectionName, $job, $maxTries)
{
    $maxTries = ! is_null($job->maxTries()) ? $job->maxTries() :
 $maxTries;

    if ($maxTries === 0 || $job->attempts() <= $maxTries) {
        return;
    }

    $this->failJob($connectionName, $job, $e = new MaxAttemptsEx
ceededException(
        'A queued job has been attempted too many times. The job
 may have previously timed out.'
    ));

    throw $e;
}

public function maxTries()
{
    return array_get($this->payload(), 'maxTries');
}

public function attempts()
{
    return Arr::get($this->decoded, 'attempts') + 1;
}

protected function failJob($connectionName, $job, $e)
{
    return FailingJob::handle($connectionName, $job, $e);
}
```

当遇到重试次数大于限制的任务，`work` 进程就会调用 `FailingJob`：

```
protected function failJob($connectionName, $job, $e)
{
    return FailingJob::handle($connectionName, $job, $e);
```

```
}

public static function handle($connectionName, $job, $e = null)
{
    $job->markAsFailed();

    if ($job->isDeleted()) {
        return;
    }

    try {
        $job->delete();

        $job->failed($e);
    } finally {
        static::events()->fire(new JobFailed(
            $connectionName, $job, $e ?: new ManuallyFailedExcep
tion
        ));
    }
}

public function markAsFailed()
{
    $this->failed = true;
}

public function delete()
{
    parent::delete();

    $this->redis->deleteReserved($this->queue, $this);
}

public function isDeleted()
{
    return $this->deleted;
}
```

`FailingJob` 会标记当前任务 `failed` 、 `deleted` ，并且会将当前任务移除 `reserved` 队列，不会再重试：

```
public function deleteReserved($queue, $job)
{
    $this->getConnection()->zrem($this->getQueue($queue).':reser
ved', $job->getReservedJob());
}
```

`FailingJob` 还会调用 `RedisJob` 的 `failed` 函数，并且触发 `JobFailed` 事件：

```
public function failed($e)
{
    $this->markAsFailed();

    $payload = $this->payload();

    list($class, $method) = JobName::parse($payload['job']);

    if (method_exists($this->instance = $this->resolve($class),
'failed')) {
        $this->instance->failed($payload['data'], $e);
    }
}
```

程序会解析 `job` 类，我们先前在 `redis` 中已经存储了：

```
[
    'job' => 'Illuminate\Queue\CallQueuedHandler@call',
    'maxTries' => isset($job->tries) ? $job->tries : null,
    'timeout' => isset($job->timeout) ? $job->timeout : null,
    'data' => [
        'commandName' => get_class($job),
        'command' => serialize(clone $job),
    ],
];
```

我们接着看 `failed` 函数：

```
public function failed(array $data, $e)
{
    $command = unserialize($data['command']);

    if (method_exists($command, 'failed')) {
        $command->failed($e);
    }
}
```

可以看到，最后程序调用了任务类的 `failed` 函数。

异常处理

当任务遇到异常的时候，程序仍然会判断当前任务的重试次数，如果本次任务的重试次数已经大于或等于限制，那么就会停止重试，标记为失败；否则就会重新放入队列，记录日志。

```
protected function handleJobException($connectionName, $job, Wor
kerOptions $options, $e)
{
    try {
        $this->markJobAsFailedIfWillExceedMaxAttempts(
            $connectionName, $job, (int) $options->maxTries, $e
        );

        $this->raiseExceptionOccurredJobEvent(
            $connectionName, $job, $e
        );
    } finally {
        if (! $job->isDeleted()) {
            $job->release($options->delay);
        }
    }

    throw $e;
}
```

```
protected function markJobAsFailedIfWillExceedMaxAttempts($conne
ctionName, $job, $maxTries, $e)
{
    $maxTries = ! is_null($job->maxTries()) ? $job->maxTries() :
 $maxTries;

    if ($maxTries > 0 && $job->attempts() >= $maxTries) {
        $this->failJob($connectionName, $job, $e);
    }
}

public function release($delay = 0)
{
    parent::release($delay);

    $this->redis->deleteAndRelease($this->queue, $this, $delay);
}

public function deleteAndRelease($queue, $job, $delay)
{
    $queue = $this->getQueue($queue);

    $this->getConnection()->eval(
        LuaScripts::release(), 2, $queue.':delayed', $queue.':re
served',
        $job->getReservedJob(), $this->availableAt($delay)
    );
}
```

一旦任务出现异常错误。那么该任务将会立刻从 `reserved` 队列放入 `delayed` 队列，并且抛出异常，抛出异常后，程序会将其记录在日志中。

```
public static function release()
{
    return <<<'LUA'
        -- Remove the job from the current queue...
        redis.call('zrem', KEYS[2], ARGV[1])

        -- Add the job onto the "delayed" queue...
        redis.call('zadd', KEYS[1], ARGV[2], ARGV[1])

        return true
    LUA;
}
```

任务的运行

任务的运行首先会调用 `CallQueuedHandler` 的 `call` 函数：

```
public function fire()
{
    $payload = $this->payload();

    list($class, $method) = JobName::parse($payload['job']);

    with($this->instance = $this->resolve($class))->{$method}($t
his, $payload['data']);
}

public function call(Job $job, array $data)
{
    $command = $this->setJobInstanceIfNecessary(
        $job, unserialize($data['command'])
    );

    $this->dispatcher->dispatchNow(
        $command, $handler = $this->resolveHandler($job, $comman
d)
    );

    if (! $job->isDeletedOrReleased()) {
        $job->delete();
    }
}
```

`setJobInstanceIfNecessary` 函数用于为任务类的 `trait` : `InteractsWithQueue` 的设置任务类：

```
protected function setJobInstanceIfNecessary(Job $job, $instance)

{
    if (in_array(InteractsWithQueue::class, class_uses_recursive
(get_class($instance)))) {
        $instance->setJob($job);
    }

    return $instance;
}

trait InteractsWithQueue
{
    public function setJob(JobContract $job)
    {
        $this->job = $job;

        return $this;
    }
}
```

接着任务的运行就要交给 `dispatch` :

```
public function dispatchNow($command, $handler = null)
{
    if ($handler || $handler = $this->getCommandHandler($command
)) {
        $callback = function ($command) use ($handler) {
            return $handler->handle($command);
        };
    } else {
        $callback = function ($command) {
            return $this->container->call([$command, 'handle']);
        };
    }

    return $this->pipeline->send($command)->through($this->pipes
)->then($callback);
}

public function getCommandHandler($command)
{
    if ($this->hasCommandHandler($command)) {
        return $this->container->make($this->handlers[get_class(
$command)]);
    }

    return false;
}

public function hasCommandHandler($command)
{
    return array_key_exists(get_class($command), $this->handlers
);
}
```

如果不对 `dispatcher` 类进行任何 `map` 函数设置，`getCommandHandler` 将会返回 `null`，此时就会调用任务类的 `handle` 函数，进行具体的业务逻辑。

任务结束后，就会调用 `delete` 函数：

```
public function delete()
{
    parent::delete();

    $this->redis->deleteReserved($this->queue, $this);
}

public function deleteReserved($queue, $job)
{
    $this->getConnection()->zrem($this->getQueue($queue).':reserved', $job->getReservedJob());
}
```

这样，运行成功的任务会从 `reserved` 中删除。

Laravel Broadcast——广播系统源码剖析

前言

在现代的 `web` 应用程序中， `WebSockets` 被用来实现需要实时、即时更新的接口。当服务器上的数据被更新后，更新信息将通过 `WebSocket` 连接发送到客户端等待处理。相比于不停地轮询应用程序， `WebSocket` 是一种更加可靠和高效的选择。

我们先用一个电子商务网站作为例子来概览一下事件广播。当用户在查看自己的订单时，我们不希望他们必须通过刷新页面才能看到状态更新。我们希望一旦有更新时就主动将更新信息广播到客户端。

`laravel` 的广播系统和队列系统类似，需要两个进程协作，一个是 `laravel` 的 `web` 后台系统，另一个是 `Socket.IO` 服务器系统。具体的流程是页面加载时，网页 `js` 程序 `Laravel Echo` 与 `Socket.IO` 服务器建立连接， `laravel` 发起通过驱动发布广播， `Socket.IO` 服务器接受广播内容，对连接的客户端网页推送信息，以达到网页实时更新的目的。

`laravel` 发起广播的方式有两种， `redis` 与 `pusher` 。对于 `redis` 来说，需要支持 `Socket.IO` 服务器系统，官方推荐 `nodejs` 为底层的 `tlaverdure/laravel-echo-server` 。对于 `pusher` 来说，该第三方服务包含了驱动与 `Socket.IO` 服务器。

本文将会介绍 `redis` 为驱动的广播源码，由于 `laravel-echo-server` 是 `nodejs` 编写，本文也无法介绍 `Socket.IO` 方面的内容。

广播系统服务的启动

和其他服务类似，广播系统服务的注册实质上就是对 `Ioc` 容器注册门面类，广播系统的门面类是 `BroadcastManager` :

```
class BroadcastServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->app->singleton(BroadcastManager::class, function
($app) {
            return new BroadcastManager($app);
        });

        $this->app->singleton(BroadcasterContract::class, functi
on ($app) {
            return $app->make(BroadcastManager::class)->connecti
on();
        });

        $this->app->alias(
            BroadcastManager::class, BroadcastingFactory::class
        );
    }
}
```

除了注册 `BroadcastManager`，`BroadcastServiceProvider` 还进行了广播驱动的启动：

```
public function connection($driver = null)
{
    return $this->driver($driver);
}

public function driver($name = null)
{
    $name = $name ?: $this->getDefaultDriver();

    return $this->drivers[$name] = $this->get($name);
}

protected function get($name)
{
    return isset($this->drivers[$name]) ? $this->drivers[$name]
```

```
: $this->resolve($name);
}

protected function resolve($name)
{
    $config = $this->getConfig($name);

    if (is_null($config)) {
        throw new InvalidArgumentException("Broadcaster [{$name}
] is not defined.");
    }

    if (isset($this->customCreators[$config['driver']])) {
        return $this->callCustomCreator($config);
    }

    $driverMethod = 'create'.ucfirst($config['driver']).'Driver'
;

    if (! method_exists($this, $driverMethod)) {
        throw new InvalidArgumentException("Driver [{$config['dr
iver']}] is not supported.");
    }

    return $this->{$driverMethod}($config);
}

protected function createRedisDriver(array $config)
{
    return new RedisBroadcaster(
        $this->app->make('redis'), Arr::get($config, 'connection'
)
    );
}
```

广播信息的发布

广播信息的发布与事件的发布大致相同，要告知 `Laravel` 一个给定的事件是广播类型，只需在事件类中实现 `Illuminate\Contracts\Broadcasting\ShouldBroadcast` 接口即可。该接口已经被导入到所有由框架生成的事件类中，所以可以很方便地将它添加到自己的事件中。

`ShouldBroadcast` 接口要求你实现一个方法：`broadcastOn` . `broadcastOn` 方法返回一个频道或一个频道数组，事件会被广播到这些频道。频道必须是 `Channel` 、`PrivateChannel` 或 `PresenceChannel` 的实例。`Channel` 实例表示任何用户都可以订阅的公开频道，而 `PrivateChannels` 和 `PresenceChannels` 则表示需要 频道授权 的私有频道：

```
class ServerCreated implements ShouldBroadcast
{
    use SerializesModels;

    public $user;

    //默认情况下，每一个广播事件都被添加到默认的队列上，默认的队列连接在 queue.php 配置文件中指定。可以通过在事件类中定义一个 broadcastQueue 属性来自定义广播器使用的队列。该属性用于指定广播使用的队列名称：
    public $broadcastQueue = 'your-queue-name';

    public function __construct(User $user)
    {
        $this->user = $user;
    }

    public function broadcastOn()
    {
        return new PrivateChannel('user.'.$this->user->id);
    }

    //Laravel 默认会使用事件的类名作为广播名称来广播事件，自定义：
    public function broadcastAs()
    {
        return 'server.created';
    }
```

```
    //想更细粒度地控制广播数据:
    public function broadcastWith()
    {
        return ['id' => $this->user->id];
    }

    //有时，想在给定条件为 true ，才广播事件:
    public function broadcastWhen()
    {
        return $this->value > 100;
    }
}
```

然后，只需要像平时那样触发事件。一旦事件被触发，一个队列任务会自动广播事件到你指定的广播驱动器上。

当一个事件被广播时，它所有的 `public` 属性会自动被序列化为广播数据，这允许你在你的 `JavaScript` 应用中访问事件的公有数据。因此，举个例子，如果你的事件有一个公有的 `$user` 属性，它包含了一个 `Elouqent` 模型，那么事件的广播数据会是：

```
{
    "user": {
        "id": 1,
        "name": "Patrick Stewart"
        ...
    }
}
```

广播发布的源码

广播的发布与事件的触发是一体的，具体的流程我们已经在 `event` 的源码中介绍清楚了，现在我们来看唯一的不同：

```
public function dispatch($event, $payload = [], $halt = false)
    {
        list($event, $payload) = $this->parseEventAndPayload(
            $event, $payload
        );

        if ($this->shouldBroadcast($payload)) {
            $this->broadcastEvent($payload[0]);
        }

        ...
    }

    protected function shouldBroadcast(array $payload)
    {
        return isset($payload[0]) && $payload[0] instanceof Shou
ldBroadcast;
    }

    protected function broadcastEvent($event)
    {
        $this->container->make(BroadcastFactory::class)->queue($
event);
    }
```

可见，关键之处在于 `BroadcastManager` 的 `quene` 方法：

```
public function queue($event)
{
    $connection = $event instanceof ShouldBroadcastNow ? 'sync'
: null;

    if (is_null($connection) && isset($event->connection)) {
        $connection = $event->connection;
    }

    $queue = null;

    if (isset($event->broadcastQueue)) {
        $queue = $event->broadcastQueue;
    } elseif (isset($event->queue)) {
        $queue = $event->queue;
    }

    $this->app->make('queue')->connection($connection)->pushOn(
        $queue, new BroadcastEvent(clone $event)
    );
}
```

可见， `quene` 方法将广播事件包装为事件类，并且通过队列发布，我们接下来看这个事件类的处理：

```
class BroadcastEvent implements ShouldQueue
{
    public function handle(Broadcaster $broadcaster)
    {
        $name = method_exists($this->event, 'broadcastAs')
                ? $this->event->broadcastAs() : get_class($this-
>event);

        $broadcaster->broadcast(
            array_wrap($this->event->broadcastOn()), $name,
            $this->getPayloadFromEvent($this->event)
        );
    }
```

```
    protected function getPayloadFromEvent($event)
    {
        if (method_exists($event, 'broadcastWith')) {
            return array_merge(
                $event->broadcastWith(), ['socket' => data_get($
event, 'socket')]
            );
        }

        $payload = [];

        foreach ((new ReflectionClass($event))->getProperties(Re
flectionProperty::IS_PUBLIC) as $property) {
            $payload[$property->getName()] = $this->formatProper
ty($property->getValue($event));
        }

        return $payload;
    }

    protected function formatProperty($value)
    {
        if ($value instanceof Arrayable) {
            return $value->toArray();
        }

        return $value;
    }
}
```

可见该事件主要调用 `broadcaster` 的 `broadcast` 方法，我们这里讲 `redis` 的发布：

```
class RedisBroadcaster extends Broadcaster
{
    public function broadcast(array $channels, $event, array $pa
yload = [])
    {
        $connection = $this->redis->connection($this->connection
);

        $payload = json_encode([
            'event' => $event,
            'data' => $payload,
            'socket' => Arr::pull($payload, 'socket'),
        ]);

        foreach ($this->formatChannels($channels) as $channel) {
            $connection->publish($channel, $payload);
        }
    }
}

protected function formatChannels(array $channels)
{
    return array_map(function ($channel) {
        return (string) $channel;
    }, $channels);
}
```

`broadcast` 方法运用了 `redis` 的 `publish` 方法，对 `redis` 进行了频道的信息发布。

频道授权

对于私有频道，用户只有被授权后才能监听。实现过程是用户向 `Laravel` 应用程序发起一个携带频道名称的 `HTTP` 请求，应用程序判断该用户是否能够监听该频道。在使用 `Laravel Echo` 时，上述 `HTTP` 请求会被自动发送；尽管如此，仍然需要定义适当的路由来响应这些请求。

定义授权路由

我们可以在 Laravel 里很容易地定义路由来响应频道授权请求。

```
Broadcast::routes();
```

`Broadcast::routes` 方法会自动把它的路由放进 `web` 中间件组中；另外，如果你想对一些属性自定义，可以向该方法传递一个包含路由属性的数组

```
Broadcast::routes($attributes);
```

定义授权回调

接下来，我们需要定义真正用于处理频道授权的逻辑。这是在 `routes/channels.php` 文件中完成。在该文件中，你可以用 `Broadcast::channel` 方法来注册频道授权回调函数：

```
Broadcast::channel('order.{orderId}', function ($user, $orderId)
{
    return $user->id === Order::findOrNew($orderId)->user_id;
});
```

`channel` 方法接收两个参数：频道名称和一个回调函数，该回调通过返回 `true` 或 `false` 来表示用户是否被授权监听该频道。

所有的授权回调接收当前被认证的用户作为第一个参数，任何额外的通配符参数作为后续参数。在本例中，我们使用 `{orderId}` 占位符来表示频道名称的「ID」部分是通配符。

授权回调模型绑定

就像 `HTTP` 路由一样，频道路由也可以利用显式或隐式 路由模型绑定。例如，相比于接收一个字符串或数字类型的 `order ID` ，你也可以请求一个真正的 `Order` 模型实例:

```
Broadcast::channel('order.{order}', function ($user, Order $orde
r) {
    return $user->id === $order->user_id;
});
```

频道授权源码分析

授权路由

```
class BroadcastManager implements FactoryContract
{
    public function routes(array $attributes = null)
    {
        if ($this->app->routesAreCached()) {
            return;
        }

        $attributes = $attributes ?: ['middleware' => ['web']];

        $this->app['router']->group($attributes, function ($rout
er) {
            $router->post('/broadcasting/auth', BroadcastControl
ler::class.'@authenticate');
        });
    }
}
```

频道专门有 `Controller` 来处理授权服务：

```
class BroadcastController extends Controller
{
    public function authenticate(Request $request)
    {
        return Broadcast::auth($request);
    }
}
```

当 `Socket Io` 服务器对 `javascript` 程序推送数据的时候，首先会经过该 `controller` 进行授权验证：

```
public function auth($request)
{
    if (Str::startsWith($request->channel_name, ['private-', 'presence-']) &&
        ! $request->user()) {
        throw new HttpException(403);
    }

    $channelName = Str::startsWith($request->channel_name, 'private-')
                        ? Str::replaceFirst('private-', '', $request->channel_name)
                        : Str::replaceFirst('presence-', '', $request->channel_name);

    return parent::verifyUserCanAccessChannel(
        $request, $channelName
    );
}
```

`verifyUserCanAccessChannel` 根据频道与其绑定的闭包函数来验证该频道是否可以通过授权：

```
protected function verifyUserCanAccessChannel($request, $channel)

{
    foreach ($this->channels as $pattern => $callback) {
        if (! Str::is(preg_replace('/\{(.*?)\}/', '*', $pattern)
, $channel)) {
            continue;
        }

        $parameters = $this->extractAuthParameters($pattern, $ch
annel, $callback);

        if ($result = $callback($request->user(), ...$parameters
)) {
            return $this->validAuthenticationResponse($request,
$result);
        }
    }

    throw new HttpException(403);
}
```

由于频道的命名经常带有 `userid` 等参数，因此判断频道之前首先要把 `channels` 中的频道名转为通配符 `*`，例如 `order.{userid}` 转为 order.*，之后进行正则匹配。

`extractAuthParameters` 用于提取频道的闭包函数的参数，合并 `$request->user()` 之后调用闭包函数。

```
protected function extractAuthParameters($pattern, $channel, $ca
llback)
{
    $callbackParameters = (new ReflectionFunction($callback))->g
etParameters();

    return collect($this->extractChannelKeys($pattern, $channel)
)->reject(function ($value, $key) {
        return is_numeric($key);
    })->map(function ($value, $key) use ($callbackParameters) {
        return $this->resolveBinding($key, $value, $callbackPara
meters);
    })->values()->all();
}

protected function extractChannelKeys($pattern, $channel)
{
    preg_match('/^'.preg_replace('/\{(.*?)\}/', '(?<$1>[^\.]+)',
 $pattern).'/', $channel, $keys);

    return $keys;
}

public function validAuthenticationResponse($request, $result)
{
    if (is_bool($result)) {
        return json_encode($result);
    }

    return json_encode(['channel_data' => [
        'user_id' => $request->user()->getKey(),
        'user_info' => $result,
    ]]);
}
```

`extractChannelKeys` 用于将 `order.{userid}` 与 `order.23` 中 `userid` 和 `23` 建立 `key` 、 `value` 关联。如果 `userid` 是 `User` 的主键， `resolveBinding` 还可以为其自动进行路由模型绑定。

Laravel Passport——OAuth2 API 认证系统源码解析（上）

前言

在 Laravel 中，实现基于传统表单的登陆和授权已经非常简单，但是如何满足 API 场景下的授权需求呢？在 API 场景里通常通过令牌来实现用户授权，而非维护请求之间的 Session 状态。在 Laravel 项目中使用 Passport 可以轻而易举地实现 API 授权认证，Passport 可以在几分钟之内为你的应用程序提供完整的 OAuth2 服务端实现。

首先我们可以先了解一下 OAuth2：理解OAuth 2.0

可以看出来，OAuth2 的授权模式分为 4 种，相应的 Passport 的授权模式也是 4 中。下面，我们就会逐一进行源码分析。

Passport 服务的注册启动

```
class PassportServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->registerAuthorizationServer();

        $this->registerResourceServer();

        $this->registerGuard();
    }
}
```

我们知道 `OAuth2` 大致由 `客户` 、 `客户端` 、 `认证服务器` 、 `资源服务器` 等构成。 在这里，我们扮演着 `认证服务器` 与 `资源服务器` 的角色。

认证服务器注册

```
protected function registerAuthorizationServer()
{
    $this->app->singleton(AuthorizationServer::class, function ()
 {
        return tap($this->makeAuthorizationServer(), function ($
server) {
            $server->enableGrantType(
                $this->makeAuthCodeGrant(), Passport::tokensExpi
reIn()
            );

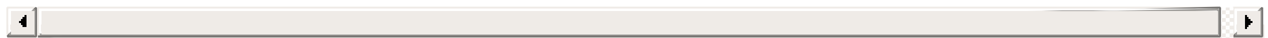
            $server->enableGrantType(
                $this->makeRefreshTokenGrant(), Passport::tokens
ExpireIn()
            );

            $server->enableGrantType(
                $this->makePasswordGrant(), Passport::tokensExpi
reIn()
            );

            $server->enableGrantType(
                new PersonalAccessGrant, new DateInterval('P1Y')
            );

            $server->enableGrantType(
                new ClientCredentialsGrant, Passport::tokensExpi
reIn()
            );

            if (Passport::$implicitGrantEnabled) {
                $server->enableGrantType(
                    $this->makeImplicitGrant(), Passport::tokens
ExpireIn()
                );
            }
        });
    });
}
```



`AuthorizationServer` 认证服务器是 `League OAuth2 server` 的一个类，是 `League` 关于 `OAuth2` 的实现类。这个认证服务器类需要 5 个参数，分别代表 `客户端` 、 `token 令牌` 、 `scope 作用范围` 、 `加密私钥` 、 `加密 key` 。

```
class AuthorizationServer implements EmitterAwareInterface
{
    public function __construct(
        ClientRepositoryInterface $clientRepository,
        AccessTokenRepositoryInterface $accessTokenRepository,
        ScopeRepositoryInterface $scopeRepository,
        $privateKey,
        $encryptionKey,
        ResponseTypeInterface $responseType = null
    ) {
        $this->clientRepository = $clientRepository;
        $this->accessTokenRepository = $accessTokenRepository;
        $this->scopeRepository = $scopeRepository;

        if ($privateKey instanceof CryptKey === false) {
            $privateKey = new CryptKey($privateKey);
        }
        $this->privateKey = $privateKey;
        $this->encryptionKey = $encryptionKey;
        $this->responseType = $responseType;
    }
}
```

这些不同的 `Repository` 均是各个接口类，这些类规定了各个部分的功能。 `Passport` 实现了上述几个接口类：

```
public function makeAuthorizationServer()
{
    return new AuthorizationServer(
        $this->app->make(Bridge\ClientRepository::class),
        $this->app->make(Bridge\AccessTokenRepository::class),
        $this->app->make(Bridge\ScopeRepository::class),
        $this->makeCryptKey('oauth-private.key'),
        app('encrypter')->getKey()
    );
}

protected function makeCryptKey($key)
{
    return new CryptKey(
        'file://'.Passport::keyPath($key),
        null,
        false
    );
}
```

`oauth-private.key` 这个私钥由 `php artisan passport:keys` 命令生成。`encrypter` 的加密 `key` 是 `.env` 文件的 `key` 属性。

构建认证服务器之后，还要对认证服务器注册授权方式。`Passport` 的授权方式有传统的 `OAuth2` ：`授权码模式` 、`密码模式` 、`隐性模式` 、`客户端模式` ，还有 `刷新令牌模式` 、`个人授权模式` 等。

```
protected function makeAuthCodeGrant()
{
    return tap($this->buildAuthCodeGrant(), function ($grant) {
        $grant->setRefreshTokenTTL(Passport::refreshTokensExpire
In());
    });
}

protected function buildAuthCodeGrant()
{
    return new AuthCodeGrant(
        $this->app->make(Bridge\AuthCodeRepository::class),
        $this->app->make(Bridge\RefreshTokenRepository::class),
        new DateInterval('PT10M')
    );
}
```

资源服务器注册

类似的, `ResourceServer` 也是 `League` 的资源服务器类：

```
protected function registerResourceServer()
{
    $this->app->singleton(ResourceServer::class, function () {
        return new ResourceServer(
            $this->app->make(Bridge\AccessTokenRepository::class
),
            $this->makeCryptKey('oauth-public.key')
        );
    });
}
```

guard 注册

当我们已经构建好 `Passport` 服务之后，我们只要利用中间件 `Auth:api` 就可以利用 `Passport` 验证 `api` 的合法性。具体的原理是 中间件 `Auth` 的参数 `api` 是指定 `guard` 的名称，例如 `web` 、 `api` ,如果调用的是 `api` 的

`guard` 那么就会创建相应的 `passport` 驱动器：

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],

    'api' => [
        'driver' => 'passport',
        'provider' => 'users',
    ],
],
```

而 `passport` 的 `guard` 驱动器就是这个 `TokenGuard`：

```
protected function registerGuard()
{
    Auth::extend('passport', function ($app, $name, array $confi
g) {
        return tap($this->makeGuard($config), function ($guard)
{
            $this->app->refresh('request', $guard, 'setRequest')
;
        });
    });
}

protected function makeGuard(array $config)
{
    return new RequestGuard(function ($request) use ($config) {
        return (new TokenGuard(
            $this->app->make(ResourceServer::class),
            Auth::createUserProvider($config['provider']),
            $this->app->make(TokenRepository::class),
            $this->app->make(ClientRepository::class),
            $this->app->make('encrypter')
        ))->user($request);
    }, $this->app['request']);
}
```

授权码模式

授权码模式大概分为 5 个步骤：

- 第三方 向我们的服务器申请创建客户端。
- 用户打开客户端以后，客户端会跳转到我们的网站授权页面要求用户给予授权。
- 用户同意给予客户端授权，我们将会返回 `授权码` 。
- 客户端使用上一步获得的授权，向认证服务器申请令牌。
- 客户端使用令牌，向资源服务器申请获取资源。

为何授权码模式需要如此设置步骤可以查看：Why is there an “Authorization Code” flow in OAuth2 when “Implicit” flow works so well?、OAuth2疑问解答

创建客户端

在创建客户端这一步骤，第三方需要提供客户端名称与客户端的 `redirect` ：

```
const data = {
    name: 'Client Name',
    redirect: 'http://example.com/callback'
};

axios.post('/oauth/clients', data)
    .then(response => {
        console.log(response.data);
    })
    .catch (response => {
        // List errors on response...
    });
```

我们在创建成功之后，会返回此客户端的 `ID` 和密钥。这两个东西十分重要，是后面几个步骤必要的参数。

```
public function forClients()
{
    $this->router->group(['middleware' => ['web', 'auth']], func
tion ($router) {
        $router->get('/clients', [
            'uses' => 'ClientController@forUser',
        ]);

        $router->post('/clients', [
            'uses' => 'ClientController@store',
        ]);

        $router->put('/clients/{client_id}', [
            'uses' => 'ClientController@update',
        ]);

        $router->delete('/clients/{client_id}', [
            'uses' => 'ClientController@destroy',
        ]);
```

```
    });
}

public function store(Request $request)
{
    $this->validation->make($request->all(), [
        'name' => 'required|max:255',
        'redirect' => 'required|url',
    ])->validate();

    return $this->clients->create(
        $request->user()->getKey(), $request->name, $request->re
direct
    )->makeVisible('secret');
}

public function create($userId, $name, $redirect, $personalAcces
s = false, $password = false)
{
    $client = (new Client)->forceFill([
        'user_id' => $userId,
        'name' => $name,
        'secret' => str_random(40),
        'redirect' => $redirect,
        'personal_access_client' => $personalAccess,
        'password_client' => $password,
        'revoked' => false,
    ]);

    $client->save();

    return $client;
}
```

跳转授权页面

客户端创建之后，开发者会使用此客户端的 ID 和密钥来请求授权代码，并从应用程序访问令牌。首先，接入应用的用户向你应用程序的 /oauth/authorize 路由发出重定向请求，

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'code',
        'scope' => '',
    ]);

    return redirect('http://your-app.com/oauth/authorize?'.$quer
y);
});
```

这个链接会访问我们的授权路由，我们的服务器会验证上面的四个参数，考察是否存在这个第三方客户端，如果验证通过，将会渲染出我们的授权页面。

```
public function forAuthorization()
{
    $this->router->group(['middleware' => ['web', 'auth']], func
tion ($router) {
        $router->get('/authorize', [
            'uses' => 'AuthorizationController@authorize',
        ]);

        $router->post('/authorize', [
            'uses' => 'ApproveAuthorizationController@approve',
        ]);

        $router->delete('/authorize', [
            'uses' => 'DenyAuthorizationController@deny',
        ]);
    });
}

public function authorize(ServerRequestInterface $psrRequest,
                          Request $request,
                          ClientRepository $clients,
                          TokenRepository $tokens)
{
    return $this->withErrorHandling(function () use ($psrRequest
```

```
, $request, $clients, $tokens) {
        $authRequest = $this->server->validateAuthorizationReque
st($psrRequest);

        $scopes = $this->parseScopes($authRequest);

        $token = $tokens->findValidToken(
            $user = $request->user(),
            $client = $clients->find($authRequest->getClient()->
getIdentifier())
        );

        if ($token && $token->scopes === collect($scopes)->pluck(
'id')->all()) {
            return $this->approveRequest($authRequest, $user);
        }

        $request->session()->put('authRequest', $authRequest);

        return $this->response->view('passport::authorize', [
            'client' => $client,
            'user' => $user,
            'scopes' => $scopes,
            'request' => $request,
        ]);
    });
}
```

这里最关键的就是 `validateAuthorizationRequest` 这个函数：

```
public function validateAuthorizationRequest(ServerRequestInterface $request)
{
    foreach ($this->enabledGrantTypes as $grantType) {
        if ($grantType->canRespondToAuthorizationRequest($request)) {
            return $grantType->validateAuthorizationRequest($request);
        }
    }

    throw OAuthServerException::unsupportedGrantType();
}
```

`canRespondToAuthorizationRequest` 用于验证授权模式与参数的 `response_type` 是否符合。如果确认授权模式正确，那么接下来就会继续验证以下几项：

- 客户端 id
- redirect 重定向地址
- scopes 授权范围
- state 客户端状态

客户端

```
public function validateAuthorizationRequest(ServerRequestInterf
ace $request)
{
    $clientId = $this->getQueryStringParameter(
        'client_id',
        $request,
        $this->getServerParameter('PHP_AUTH_USER', $request)
    );
    if (is_null($clientId)) {
        throw OAuthServerException::invalidRequest('client_id');
    }

    $client = $this->clientRepository->getClientEntity(
        $clientId,
        $this->getIdentifier(),
        null,
        false
    );

    if ($client instanceof ClientEntityInterface === false) {
        $this->getEmitter()->emit(new RequestEvent(RequestEvent:
:CLIENT_AUTHENTICATION_FAILED, $request));
        throw OAuthServerException::invalidClient();
    }

    ...
}

public function getIdentifier()
{
    return 'authorization_code';
}
```

客户端的验证主要是利用请求中的参数 `client_id` ，我们会从表 `oauth_clients` 的表中按照 `client_id` 来取出数据库记录：

```
public function getClientEntity($clientIdentifier, $grantType,
                                $clientSecret = null, $mustV
alidateSecret = true)
```

```
{
    $record = $this->clients->findActive($clientIdentifier);

    if (! $record || ! $this->handlesGrant($record, $grantType))
 {
        return;
    }

    $client = new Client(
        $clientIdentifier, $record->name, $record->redirect
    );

    if ($mustValidateSecret &&
        ! hash_equals($record->secret, (string) $clientSecret))
{
        return;
    }

    return $client;
}

public function findActive($id)
{
    $client = $this->find($id);

    return $client && ! $client->revoked ? $client : null;
}

protected function handlesGrant($record, $grantType)
{
    switch ($grantType) {
        case 'authorization_code':
            return ! $record->firstParty();
        case 'personal_access':
            return $record->personal_access_client;
        case 'password':
            return $record->password_client;
        default:
            return true;
    }
```

```
}
```

在表 `oauth_clients` 中还有两个字段 `personal_access` 、 `password` ，对于授权码模式来说这两个字段都要求为 0。

重定向地址

```
public function validateAuthorizationRequest(ServerRequestInterf
ace $request)
{
    ...

    $redirectUri = $this->getQueryStringParameter('redirect_uri'
, $request);
    if ($redirectUri !== null) {
        if (
            is_string($client->getRedirectUri())
            && (strcmp($client->getRedirectUri(), $redirectUri)
!== 0)
        ) {
            $this->getEmitter()->emit(new RequestEvent(RequestEv
ent::CLIENT_AUTHENTICATION_FAILED, $request));
            throw OAuthServerException::invalidClient();
        } elseif (
            is_array($client->getRedirectUri())
            && in_array($redirectUri, $client->getRedirectUri())
 === false
        ) {
            $this->getEmitter()->emit(new RequestEvent(RequestEv
ent::CLIENT_AUTHENTICATION_FAILED, $request));
            throw OAuthServerException::invalidClient();
        }
    } elseif (is_array($client->getRedirectUri()) && count($clie
nt->getRedirectUri()) !== 1
        || empty($client->getRedirectUri())
    ) {
        $this->getEmitter()->emit(new RequestEvent(RequestEvent:
:CLIENT_AUTHENTICATION_FAILED, $request));
        throw OAuthServerException::invalidClient();
    }

    ...

}
```

这部分验证参数中的 `redirect_uri` 是否与数据库中的重定向地址是否一致。

授权作用域

授权作用域可以让 API 客户端在请求账户授权时请求特定的权限。例如，如果你正在构建电子商务应用程序，并不是所有接入的 API 应用都需要下订单的功能。你可以让接入的 API 应用只被允许授权访问订单发货状态。换句话说，作用域允许应用程序的用户限制第三方应用程序执行的操作。

你可以在 AuthServiceProvider 的 boot 方法中使用 Passport::tokensCan 方法来定义 API 的作用域。tokensCan 方法接受一个作用域名称、描述的数组作为参数。作用域描述将会在授权确认页中直接展示给用户，你可以将其定义为任何你需要的内容：

```
Passport::tokensCan([
    'place-orders' => 'Place orders',
    'check-status' => 'Check order status',
]);

public static function tokensCan(array $scopes)
{
    static::$scopes = $scopes;
}
```

验证授权作用域的时候，只是在 `Passport` 中验证是否存在该授权作用域：

```
public function validateAuthorizationRequest(ServerRequestInterf
ace $request)
{
    ...

    $scopes = $this->validateScopes(
        $this->getQueryStringParameter('scope', $request, $this-
>defaultScope),
        is_array($client->getRedirectUri())
            ? $client->getRedirectUri()[0]
            : $client->getRedirectUri()
    );

    ...
}
```

```
public function validateScopes($scopes, $redirectUri = null)
{
    $scopesList = array_filter(explode(self::SCOPE_DELIMITER_STR
ING, trim($scopes)), function ($scope) {
        return !empty($scope);
    });

    $validScopes = [];

    foreach ($scopesList as $scopeItem) {
        $scope = $this->scopeRepository->getScopeEntityByIdentif
ier($scopeItem);

        if ($scope instanceof ScopeEntityInterface === false) {
            throw OAuthServerException::invalidScope($scopeItem,
 $redirectUri);
        }

        $validScopes[] = $scope;
    }

    return $validScopes;
}

public function getScopeEntityByIdentifier($identifier)
{
    if (Passport::hasScope($identifier)) {
        return new Scope($identifier);
    }
}
```

state

这个字段用于防止 csrf 攻击的，具体可以查看 ：移花接木：针对OAuth2的CSRF攻击

```
public function validateAuthorizationRequest(ServerRequestInterf
ace $request)
{
    $stateParameter = $this->getQueryStringParameter('state', $r
equest);

        $authorizationRequest = new AuthorizationRequest();
        $authorizationRequest->setGrantTypeId($this->getIdentifi
er());
        $authorizationRequest->setClient($client);
        $authorizationRequest->setRedirectUri($redirectUri);
        $authorizationRequest->setState($stateParameter);
        $authorizationRequest->setScopes($scopes);
}
```

验证结束后，接下来就会验证当前用户是否已经授权过，如果已经授权过，那么就会直接返回授权码，否则就会渲染授权页面：

```
public function authorize(ServerRequestInterface $psrRequest,
                              Request $request,
                              ClientRepository $clients,
                              TokenRepository $tokens)
{
    return $this->withErrorHandling(function () use ($psrRequest
, $request, $clients, $tokens) {
        $authRequest = $this->server->validateAuthorizationReque
st($psrRequest);

        $scopes = $this->parseScopes($authRequest);

        $token = $tokens->findValidToken(
            $user = $request->user(),
            $client = $clients->find($authRequest->getClient()->
getIdentifier())
        );

        if ($token && $token->scopes === collect($scopes)->pluck(
'id')->all()) {
            return $this->approveRequest($authRequest, $user);
        }

        $request->session()->put('authRequest', $authRequest);

        return $this->response->view('passport::authorize', [
            'client' => $client,
            'user' => $user,
            'scopes' => $scopes,
            'request' => $request,
        ]);
    });
}
```

验证用户的是否授权首先是查看授权作用域是否与数据库保持一致。由于授权作用域与 `token` 相互关联，并非与客户端相互关联，所以 `scopes` 没有在 `oauth_clients` 表中，而是在 `oauth_access_tokens` 这个表中。

```
protected function parseScopes($authRequest)
{
    return Passport::scopesFor(
        collect($authRequest->getScopes())->map(function ($scope)
 {
            return $scope->getIdentifier();
        })->all()
    );
}

public static function scopesFor(array $ids)
{
    return collect($ids)->map(function ($id) {
        if (isset(static::$scopes[$id])) {
            return new Scope($id, static::$scopes[$id]);
        }

        return;
    })->filter()->values()->all();
}
```

可以看到，作用域的 `identifier` 就是 `Scope` 的 `id` 。

获取已授权 token

token 的获取主要是利用 `client_id` 与 `user_id` 在表 `oauth_access_tokens` 中查询符合条件的 `token` 。

```
public function findValidToken($user, $client)
{
    return $client->tokens()
                  ->whereUserId($user->getKey())
                  ->whereRevoked(0)
                  ->where('expires_at', '>', Carbon::now())
                  ->latest('expires_at')
                  ->first();
}

public function tokens()
{
    return $this->hasMany(Token::class, 'client_id');
}
```

在获取到有效的 `token` 之后，并且 `token` 的作用域符合请求参数，就会立即返回，不需要用户的重复授权：

```
protected function approveRequest($authRequest, $user)
{
    $authRequest->setUser(new User($user->getKey()));

    $authRequest->setAuthorizationApproved(true);

    return $this->convertResponse(
        $this->server->completeAuthorizationRequest($authRequest
, new Psr7Response)
    );
}
```

授权成功

用户点击确认按钮，授权成功之后，服务器就会跳转到客户端预设的 `redirecturi` ，并且携带授权码等一系列参数

```
$router->post('/authorize', [
    'uses' => 'ApproveAuthorizationController@approve',
]);


public function approve(Request $request)
{
    return $this->withErrorHandling(function () use ($request) {
        $authRequest = $this->getAuthRequestFromSession($request
);

        return $this->convertResponse(
            $this->server->completeAuthorizationRequest($authReq
uest, new Psr7Response)
        );
    });
}
```

`completeAuthorizationRequest` 是授权服务器的重要步骤：

```
public function completeAuthorizationRequest(AuthorizationReques
t $authRequest, ResponseInterface $response)
{
    return $this->enabledGrantTypes[$authRequest->getGrantTypeId
()]
        ->completeAuthorizationRequest($authRequest)
        ->generateHttpResponse($response);
}


public function completeAuthorizationRequest(AuthorizationReques
t $authorizationRequest)
{
    if ($authorizationRequest->getUser() instanceof UserEntityIn
terface === false) {
        throw new \LogicException('An instance of UserEntityInte
rface should be set on the AuthorizationRequest');
    }
```

```
    $finalRedirectUri = ($authorizationRequest->getRedirectUri()
 === null)
        ? is_array($authorizationRequest->getClient()->getRedire
ctUri())
            ? $authorizationRequest->getClient()->getRedirectUri
()[0]
            : $authorizationRequest->getClient()->getRedirectUri
()
        : $authorizationRequest->getRedirectUri();

    // The user approved the client, redirect them back with an
auth code
    if ($authorizationRequest->isAuthorizationApproved() === true
) {
        $authCode = $this->issueAuthCode(
            $this->authCodeTTL,
            $authorizationRequest->getClient(),
            $authorizationRequest->getUser()->getIdentifier(),
            $authorizationRequest->getRedirectUri(),
            $authorizationRequest->getScopes()
        );

        $payload = [
            'client_id'             => $authCode->getClient()->g
etIdentifier(),
            'redirect_uri'          => $authCode->getRedirectUri
(),
            'auth_code_id'          => $authCode->getIdentifier(
),
            'scopes'                => $authCode->getScopes(),
            'user_id'               => $authCode->getUserIdentif
ier(),
            'expire_time'           => (new \DateTime())->add($t
his->authCodeTTL)->format('U'),
            'code_challenge'        => $authorizationRequest->ge
tCodeChallenge(),
            'code_challenge_method' => $authorizationRequest->ge
tCodeChallengeMethod(),
        ];
```

```
            $response = new RedirectResponse();
            $response->setRedirectUri(
                $this->makeRedirectUri(
                    $finalRedirectUri,
                    [
                        'code'  => $this->encrypt(
                            json_encode(
                                $payload
                            )
                        ),
                        'state' => $authorizationRequest->getState()
,
                    ]
                )
            );

            return $response;
        }

        // The user denied the client, redirect them back with an er
ror
        throw OAuthServerException::accessDenied(
            'The user denied the request',
            $this->makeRedirectUri(
                $finalRedirectUri,
                [
                    'state' => $authorizationRequest->getState(),
                ]
            )
        );
    }
```

这里最重要的就是 `issueAuthCode` 生成授权码：

```
protected function issueAuthCode(
    \DateInterval $authCodeTTL,
    ClientEntityInterface $client,
    $userIdentifier,
    $redirectUri,
    array $scopes = []
) {
    $maxGenerationAttempts = self::MAX_RANDOM_TOKEN_GENERATION_A
TTEMPTS;

    $authCode = $this->authCodeRepository->getNewAuthCode();
    $authCode->setExpiryDateTime((new \DateTime())->add($authCod
eTTL));
    $authCode->setClient($client);
    $authCode->setUserIdentifier($userIdentifier);
    $authCode->setRedirectUri($redirectUri);

    foreach ($scopes as $scope) {
        $authCode->addScope($scope);
    }

    while ($maxGenerationAttempts-- > 0) {
        $authCode->setIdentifier($this->generateUniqueIdentifier
());
        try {
            $this->authCodeRepository->persistNewAuthCode($authC
ode);

            return $authCode;
        } catch (UniqueTokenIdentifierConstraintViolationExcepti
on $e) {
            if ($maxGenerationAttempts === 0) {
                throw $e;
            }
        }
    }
}
```

其中 `generateUniqueIdentifier` 就是生成授权码的步骤，这个授权码也是表 `oauth_auth_codes` 的 `id` :

```
protected function generateUniqueIdentifier($length = 40)
{
    try {
        return bin2hex(random_bytes($length));
        // @codeCoverageIgnoreStart
    } catch (\TypeError $e) {
        throw OAuthServerException::serverError('An unexpected e
rror has occurred');
    } catch (\Error $e) {
        throw OAuthServerException::serverError('An unexpected e
rror has occurred');
    } catch (\Exception $e) {
        // If you get this message, the CSPRNG failed hard.
        throw OAuthServerException::serverError('Could not gener
ate a random string');
    }
    // @codeCoverageIgnoreEnd
}

public function persistNewAuthCode(AuthCodeEntityInterface $auth
CodeEntity)
{
    $this->database->table('oauth_auth_codes')->insert([
        'id' => $authCodeEntity->getIdentifier(),
        'user_id' => $authCodeEntity->getUserIdentifier(),
        'client_id' => $authCodeEntity->getClient()->getIdentifi
er(),
        'scopes' => $this->formatScopesForStorage($authCodeEntit
y->getScopes()),
        'revoked' => false,
        'expires_at' => $authCodeEntity->getExpiryDateTime(),
    ]);
}
```

授权码转为令牌

由于 `client_id` 是公开的，因此上一步授权码的获取理论上很容易，真正重要的是授权码转为令牌：

```
Route::get('/callback', function (Request $request) {
    $http = new GuzzleHttp\Client;

    $response = $http->post('http://your-app.com/oauth/token', [
        'form_params' => [
            'grant_type' => 'authorization_code',
            'client_id' => 'client-id',
            'client_secret' => 'client-secret',
            'redirect_uri' => 'http://example.com/callback',
            'code' => $request->code,
        ],
    ]);

    return json_decode((string) $response->getBody(), true);
});
```

这一步需要客户端提供注册时返回的密码，

```
public function forAccessTokens()
{
    $this->router->post('/token', [
        'uses' => 'AccessTokenController@issueToken',
        'middleware' => 'throttle',
    ]);
}

public function issueToken(ServerRequestInterface $request)
{
    return $this->withErrorHandling(function () use ($request) {
        return $this->convertResponse(
            $this->server->respondToAccessTokenRequest($request,
new Psr7Response)
        );
    });
}
```

这一步需要验证的东西非常繁多，我们分部分来看：

客户端验证

客户端验证主要校验 `client_id` 、 `client_secret` 、 `redirect_uri` ：

```
public function respondToAccessTokenRequest(
    ServerRequestInterface $request,
    ResponseTypeInterface $responseType,
    \DateInterval $accessTokenTTL
) {

    $client = $this->validateClient($request);

    ...

}

protected function validateClient(ServerRequestInterface $reques
t)
{
    list($basicAuthUser, $basicAuthPassword) = $this->getBasicAu
thCredentials($request);

    $clientId = $this->getRequestParameter('client_id', $request
, $basicAuthUser);

    // If the client is confidential require the client secret
    $clientSecret = $this->getRequestParameter('client_secret',
$request, $basicAuthPassword);

    $client = $this->clientRepository->getClientEntity(
        $clientId,
        $this->getIdentifier(),
        $clientSecret,
        true
    );

    $redirectUri = $this->getRequestParameter('redirect_uri', $r
equest, null);
```

```
    return $client;
}

protected function getBasicAuthCredentials(ServerRequestInterfac
e $request)
{
    if (!$request->hasHeader('Authorization')) {
        return [null, null];
    }

    $header = $request->getHeader('Authorization')[0];
    if (strpos($header, 'Basic ') !== 0) {
        return [null, null];
    }

    if (!($decoded = base64_decode(substr($header, 6)))) {
        return [null, null];
    }

    if (strpos($decoded, ':') === false) {
        return [null, null]; // HTTP Basic header without colon
isn't valid
    }

    return explode(':', $decoded, 2);
}
```

验证授权码

客户端的密码验证通过后，就会开始验证授权码，授权码的验证主要涉及 `expire_time` 、 `client_id` 、 `auth_code_id` :

```
public function respondToAccessTokenRequest(
    ServerRequestInterface $request,
    ResponseTypeInterface $responseType,
    \DateInterval $accessTokenTTL
) {
```

```
    ...

    $encryptedAuthCode = $this->getRequestParameter('code', $req
uest, null);

    if ($encryptedAuthCode === null) {
        throw OAuthServerException::invalidRequest('code');
    }

    try {
        $authCodePayload = json_decode($this->decrypt($encrypted
AuthCode));
        if (time() > $authCodePayload->expire_time) {
            throw OAuthServerException::invalidRequest('code', '
Authorization code has expired');
        }

        if ($this->authCodeRepository->isAuthCodeRevoked($authCo
dePayload->auth_code_id) === true) {
            throw OAuthServerException::invalidRequest('code', '
Authorization code has been revoked');
        }

        if ($authCodePayload->client_id !== $client->getIdentifi
er()) {
            throw OAuthServerException::invalidRequest('code', '
Authorization code was not issued to this client');
        }

        // The redirect URI is required in this request
        $redirectUri = $this->getRequestParameter('redirect_uri'
, $request, null);
        if (empty($authCodePayload->redirect_uri) === false && $
redirectUri === null) {
            throw OAuthServerException::invalidRequest('redirect
_uri');
        }

        if ($authCodePayload->redirect_uri !== $redirectUri) {
            throw OAuthServerException::invalidRequest('redirect
```

```
_uri', 'Invalid redirect URI');
        }

    } catch (\LogicException  $e) {
        throw OAuthServerException::invalidRequest('code', 'Cann
ot decrypt the authorization code');
    }
}

public function isAuthCodeRevoked($codeId)
{
    return $this->database->table('oauth_auth_codes')
                ->where('id', $codeId)->where('revoked', 1)->exi
sts();
}
```

发放令牌

令牌的发放主要是 `access_token` 、 `refresh_token` ，并且取消相关的授权码：

```
public function respondToAccessTokenRequest(
    ServerRequestInterface $request,
    ResponseTypeInterface $responseType,
    \DateInterval $accessTokenTTL
) {

    // Issue and persist access + refresh tokens
    $accessToken = $this->issueAccessToken($accessTokenTTL, $client, $authCodePayload->user_id, $scopes);
    $refreshToken = $this->issueRefreshToken($accessToken);

    // Inject tokens into response type
    $responseType->setAccessToken($accessToken);
    $responseType->setRefreshToken($refreshToken);

    // Revoke used auth code
    $this->authCodeRepository->revokeAuthCode($authCodePayload->auth_code_id);

    return $responseType;
}
```

首先需要生成 `access_token`，之后再对表 `oauth_access_tokens` 持久化 `access_token`：

```
protected function issueAccessToken(
    \DateInterval $accessTokenTTL,
    ClientEntityInterface $client,
    $userIdentifier,
    array $scopes = []
) {
    $maxGenerationAttempts = self::MAX_RANDOM_TOKEN_GENERATION_ATTEMPTS;

    $accessToken = $this->accessTokenRepository->getNewToken($client, $scopes, $userIdentifier);
    $accessToken->setClient($client);
    $accessToken->setUserIdentifier($userIdentifier);
    $accessToken->setExpiryDateTime((new \DateTime())->add($acce
```

```
ssTokenTTL));

    foreach ($scopes as $scope) {
        $accessToken->addScope($scope);
    }

    while ($maxGenerationAttempts-- > 0) {
        $accessToken->setIdentifier($this->generateUniqueIdentif
ier());
        try {
            $this->accessTokenRepository->persistNewAccessToken(
$accessToken);

            return $accessToken;
        } catch (UniqueTokenIdentifierConstraintViolationExcepti
on $e) {
            if ($maxGenerationAttempts === 0) {
                throw $e;
            }
        }
    }
}

public function getNewToken(ClientEntityInterface $clientEntity,
 array $scopes, $userIdentifier = null)
{
    return new AccessToken($userIdentifier, $scopes);
}

protected function generateUniqueIdentifier($length = 40)
{
    try {
        return bin2hex(random_bytes($length));
        // @codeCoverageIgnoreStart
    } catch (\TypeError $e) {
        throw OAuthServerException::serverError('An unexpected e
rror has occurred');
    } catch (\Error $e) {
        throw OAuthServerException::serverError('An unexpected e
rror has occurred');
```

```
    } catch (\Exception $e) {
        // If you get this message, the CSPRNG failed hard.
        throw OAuthServerException::serverError('Could not gener
ate a random string');
    }
    // @codeCoverageIgnoreEnd
}

public function persistNewAccessToken(AccessTokenEntityInterface
 $accessTokenEntity)
{
    $this->tokenRepository->create([
        'id' => $accessTokenEntity->getIdentifier(),
        'user_id' => $accessTokenEntity->getUserIdentifier(),
        'client_id' => $accessTokenEntity->getClient()->getIdent
ifier(),
        'scopes' => $this->scopesToArray($accessTokenEntity->get
Scopes()),
        'revoked' => false,
        'created_at' => new DateTime,
        'updated_at' => new DateTime,
        'expires_at' => $accessTokenEntity->getExpiryDateTime(),
    ]);

    $this->events->dispatch(new AccessTokenCreated(
        $accessTokenEntity->getIdentifier(),
        $accessTokenEntity->getUserIdentifier(),
        $accessTokenEntity->getClient()->getIdentifier()
    ));
}
```

类似地，还有生成 `refresh_token` ：

```
protected function issueRefreshToken(AccessTokenEntityInterface $accessToken)
{
    $maxGenerationAttempts = self::MAX_RANDOM_TOKEN_GENERATION_ATTEMPTS;

    $refreshToken = $this->refreshTokenRepository->getNewRefreshToken();
    $refreshToken->setExpiryDateTime((new \DateTime())->add($this->refreshTokenTTL));
    $refreshToken->setAccessToken($accessToken);

    while ($maxGenerationAttempts-- > 0) {
        $refreshToken->setIdentifier($this->generateUniqueIdentifier());
        try {
            $this->refreshTokenRepository->persistNewRefreshToken($refreshToken);

            return $refreshToken;
        } catch (UniqueTokenIdentifierConstraintViolationException $e) {
            if ($maxGenerationAttempts === 0) {
                throw $e;
            }
        }
    }
}
```

BearerToken

为了加强安全性，根据 `OAuth2` 规范，`access_token` 与 `refresh_token` 需要利用 `Bearer Token` 的方式给出，`access token` 会被转化为 `JWT`，`refresh token` 会被加密:

```
public function generateHttpResponse(ResponseInterface $response)

{
```

```
    $expireDateTime = $this->accessToken->getExpiryDateTime()->g
etTimestamp();

    $jwtAccessToken = $this->accessToken->convertToJWT($this->pr
ivateKey);

    $responseParams = [
        'token_type'   => 'Bearer',
        'expires_in'   => $expireDateTime - (new \DateTime())->g
etTimestamp(),
        'access_token' => (string) $jwtAccessToken,
    ];
I
    if ($this->refreshToken instanceof RefreshTokenEntityInterfa
ce) {
        $refreshToken = $this->encrypt(
            json_encode(
                [
                    'client_id'        => $this->accessToken->ge
tClient()->getIdentifier(),
                    'refresh_token_id' => $this->refreshToken->g
etIdentifier(),
                    'access_token_id'  => $this->accessToken->ge
tIdentifier(),
                    'scopes'           => $this->accessToken->ge
tScopes(),
                    'user_id'          => $this->accessToken->ge
tUserIdentifier(),
                    'expire_time'      => $this->refreshToken->g
etExpiryDateTime()->getTimestamp(),
                ]
            )
        );

        $responseParams['refresh_token'] = $refreshToken;
    }

    $responseParams = array_merge($this->getExtraParams($this->a
ccessToken), $responseParams);
```

```
    $response = $response
        ->withStatus(200)
        ->withHeader('pragma', 'no-cache')
        ->withHeader('cache-control', 'no-store')
        ->withHeader('content-type', 'application/json; charset=
UTF-8');

    $response->getBody()->write(json_encode($responseParams));

    return $response;
}
```

刷新令牌

如果你的应用程序发放了短期的访问令牌，用户将需要通过在发出访问令牌时提供给他们的刷新令牌来刷新其访问令牌。该申请的 `url` 与申请令牌的链接相同，仅仅 `grant_type` 不同：

```
$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'refresh_token',
        'refresh_token' => 'the-refresh-token',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'scope' => '',
    ],
]);

return json_decode((string) $response->getBody(), true);
```

```
public function respondToAccessTokenRequest(
    ServerRequestInterface $request,
    ResponseTypeInterface $responseType,
    \DateInterval $accessTokenTTL
) {
    // Validate request
    $client = $this->validateClient($request);
```

```
    $oldRefreshToken = $this->validateOldRefreshToken($request, $client->getIdentifier());
    $scopes = $this->validateScopes($this->getRequestParameter(
        'scope',
        $request,
        implode(self::SCOPE_DELIMITER_STRING, $oldRefreshToken['scopes']))
    );

    // The OAuth spec says that a refreshed access token can have the original scopes or fewer so ensure
    // the request doesn't include any new scopes
    foreach ($scopes as $scope) {
        if (in_array($scope->getIdentifier(), $oldRefreshToken['scopes']) === false) {
            throw OAuthServerException::invalidScope($scope->getIdentifier());
        }
    }

    // Expire old tokens
    $this->accessTokenRepository->revokeAccessToken($oldRefreshToken['access_token_id']);
    $this->refreshTokenRepository->revokeRefreshToken($oldRefreshToken['refresh_token_id']);

    // Issue and persist new tokens
    $accessToken = $this->issueAccessToken($accessTokenTTL, $client, $oldRefreshToken['user_id'], $scopes);
    $refreshToken = $this->issueRefreshToken($accessToken);

    // Inject tokens into response
    $responseType->setAccessToken($accessToken);
    $responseType->setRefreshToken($refreshToken);

    return $responseType;
}
```

Laravel Passport——OAuth2 API 认证系统源码解析（下）

隐式授权

隐式授权类似于授权码授权，但是它只令牌将返回给客户端而不交换授权码。这种授权最常用于无法安全存储客户端凭据的 JavaScript 或移动应用程序。通过调用 `AuthServiceProvider` 中的 `enableImplicitGrant` 方法来启用这种授权：

```
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::enableImplicitGrant();
}
```

调用上面方法开启授权后，开发者可以使用他们的客户端 ID 从应用程序请求访问令牌。接入的应用程序应该向你的应用程序的 /oauth/authorize 路由发出重定向请求，如下所示：

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'token',
        'scope' => '',
    ]);

    return redirect('http://your-app.com/oauth/authorize?'.$quer
y);
});
```

首先仍然是验证授权请求的合法性，其流程与授权码模式基本一致：

```
public function validateAuthorizationRequest(ServerRequestInterf
ace $request)
{
    $clientId = $this->getQueryStringParameter(
        'client_id',
        $request,
        $this->getServerParameter('PHP_AUTH_USER', $request)
    );
    if (is_null($clientId)) {
        throw OAuthServerException::invalidRequest('client_id');
    }

    $client = $this->clientRepository->getClientEntity(
        $clientId,
        $this->getIdentifier(),
        null,
        false
    );

    if ($client instanceof ClientEntityInterface === false) {
        $this->getEmitter()->emit(new RequestEvent(RequestEvent:
:CLIENT_AUTHENTICATION_FAILED, $request));
        throw OAuthServerException::invalidClient();
    }

    $redirectUri = $this->getQueryStringParameter('redirect_uri'
, $request);

    $scopes = $this->validateScopes(
        $this->getQueryStringParameter('scope', $request, $this-
>defaultScope),
        is_array($client->getRedirectUri())
            ? $client->getRedirectUri()[0]
            : $client->getRedirectUri()
    );

    // Finalize the requested scopes
    $finalizedScopes = $this->scopeRepository->finalizeScopes(
        $scopes,
```

```
            $this->getIdentifier(),
            $client
        );

        $stateParameter = $this->getQueryStringParameter('state', $r
equest);

        $authorizationRequest = new AuthorizationRequest();
        $authorizationRequest->setGrantTypeId($this->getIdentifier()
);
        $authorizationRequest->setClient($client);
        $authorizationRequest->setRedirectUri($redirectUri);
        $authorizationRequest->setState($stateParameter);
        $authorizationRequest->setScopes($finalizedScopes);

        return $authorizationRequest;
    }
```

接着，当用户同意授权之后，就要直接返回 `access_token`，`League OAuth2` 直接将令牌放入 `JWT` 中发送回第三方客户端,值得注意的是依据 `OAuth2` 标准，参数都是以 `location hash` 的形式返回的，间隔符是 `#`，而不是 `?`:

```
public function __construct(\DateInterval $accessTokenTTL, $quer
yDelimiter = '#')
{
    $this->accessTokenTTL = $accessTokenTTL;
    $this->queryDelimiter = $queryDelimiter;
}

public function completeAuthorizationRequest(AuthorizationReques
t $authorizationRequest)
{
    if ($authorizationRequest->getUser() instanceof UserEntityIn
terface === false) {
        throw new \LogicException('An instance of UserEntityInte
rface should be set on the AuthorizationRequest');
    }

    $finalRedirectUri = ($authorizationRequest->getRedirectUri()
```

```
 === null)
        ? is_array($authorizationRequest->getClient()->getRedire
ctUri())
            ? $authorizationRequest->getClient()->getRedirectUri
()[0]
            : $authorizationRequest->getClient()->getRedirectUri
()
        : $authorizationRequest->getRedirectUri();

    // The user approved the client, redirect them back with an
access token
    if ($authorizationRequest->isAuthorizationApproved() === true
) {
        $accessToken = $this->issueAccessToken(
            $this->accessTokenTTL,
            $authorizationRequest->getClient(),
            $authorizationRequest->getUser()->getIdentifier(),
            $authorizationRequest->getScopes()
        );

        $response = new RedirectResponse();
        $response->setRedirectUri(
            $this->makeRedirectUri(
                $finalRedirectUri,
                [
                    'access_token' => (string) $accessToken->con
vertToJWT($this->privateKey),
                    'token_type'   => 'Bearer',
                    'expires_in'   => $accessToken->getExpiryDat
eTime()->getTimestamp() - (new \DateTime())->getTimestamp(),
                    'state'        => $authorizationRequest->get
State(),
                ],
                $this->queryDelimiter
            )
        );

        return $response;
    }
```

```
    // The user denied the client, redirect them back with an er
ror
    throw OAuthServerException::accessDenied(
        'The user denied the request',
        $this->makeRedirectUri(
            $finalRedirectUri,
            [
                'state' => $authorizationRequest->getState(),
            ]
        )
    );
}
```

这个用于构建 `jwt` 的私钥就是 `oauth-private.key` ，我们知道， `jwt` 一般有三个部分组成： `header` 、 `claim` 、 `sign` , 用于 `oauth2` 的 `jwt` 中 `claim` 主要构成有：

- aud 客户端 id
- jti access_token 随机码
- iat 生成时间
- nbf 拒绝接受 jwt 时间
- exp access_token 失效时间
- sub 用户 id

具体可以参考 : JSON Web Token (JWT) draft-ietf-oauth-json-web-token-32

```
public function convertToJWT(CryptKey $privateKey)
{
    return (new Builder())
        ->setAudience($this->getClient()->getIdentifier())
        ->setId($this->getIdentifier(), true)
        ->setIssuedAt(time())
        ->setNotBefore(time())
        ->setExpiration($this->getExpiryDateTime()->getTimestamp
())
        ->setSubject($this->getUserIdentifier())
        ->set('scopes', $this->getScopes())
        ->sign(new Sha256(), new Key($privateKey->getKeyPath(),
```

```
$privateKey->getPassPhrase()))
        ->getToken();
}

public function __construct(
    Encoder $encoder = null,
    ClaimFactory $claimFactory = null
) {
    $this->encoder = $encoder ?: new Encoder();
    $this->claimFactory = $claimFactory ?: new ClaimFactory();
    $this->headers = ['typ'=> 'JWT', 'alg' => 'none'];
    $this->claims = [];
}

public function setAudience($audience, $replicateAsHeader = fals
e)
{
    return $this->setRegisteredClaim('aud', (string) $audience,
$replicateAsHeader);
}

public function setId($id, $replicateAsHeader = false)
{
    return $this->setRegisteredClaim('jti', (string) $id, $repli
cateAsHeader);
}

public function setIssuedAt($issuedAt, $replicateAsHeader = fals
e)
{
    return $this->setRegisteredClaim('iat', (int) $issuedAt, $re
plicateAsHeader);
}

public function setNotBefore($notBefore, $replicateAsHeader = fa
lse)
{
    return $this->setRegisteredClaim('nbf', (int) $notBefore, $r
eplicateAsHeader);
}
```

```
public function setExpiration($expiration, $replicateAsHeader =
false)
{
    return $this->setRegisteredClaim('exp', (int) $expiration, $
replicateAsHeader);
}

public function setSubject($subject, $replicateAsHeader = false)
{
    return $this->setRegisteredClaim('sub', (string) $subject, $
replicateAsHeader);
}

public function sign(Signer $signer, $key)
{
    $signer->modifyHeader($this->headers);

    $this->signature = $signer->sign(
        $this->getToken()->getPayload(),
        $key
    );

    return $this;
}

public function getToken()
{
    $payload = [
        $this->encoder->base64UrlEncode($this->encoder->jsonEnco
de($this->headers)),
        $this->encoder->base64UrlEncode($this->encoder->jsonEnco
de($this->claims))
    ];

    if ($this->signature !== null) {
        $payload[] = $this->encoder->base64UrlEncode($this->sign
ature);
    }
```

```
    return new Token($this->headers, $this->claims, $this->signature, $payload);
}
```

根据 JWT 的生成方法，签名部分 `signature` 是 `header` 与 `claim` 进行 `base64` 编码后再加密的结果。

客户端模式

客户端凭据授权适用于机器到机器的认证。例如，你可以在通过 API 执行维护任务中使用此授权。要使用这种授权，你首先需要在 app/Http/Kernel.php 的 routeMiddleware 变量中添加新的中间件：

```
protected $routeMiddleware = [
    'client' => CheckClientCredentials::class,
];

Route::get('/user', function(Request $request) {
    ...
})->middleware('client');
```

接下来通过向 oauth/token 接口发出请求来获取令牌:

```
$response = $guzzle->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'client_credentials',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'scope' => 'your-scope',
    ],
]);

echo json_decode((string) $response->getBody(), true);
```

客户端模式类似于授权码模式的后一部分，利用客户端 id 与客户端密码来获取 `access_token` ：

```
public function respondToAccessTokenRequest(
    ServerRequestInterface $request,
    ResponseTypeInterface $responseType,
    \DateInterval $accessTokenTTL
) {
    // Validate request
    $client = $this->validateClient($request);
    $scopes = $this->validateScopes($this->getRequestParameter('
scope', $request, $this->defaultScope));

    // Finalize the requested scopes
    $finalizedScopes = $this->scopeRepository->finalizeScopes($s
copes, $this->getIdentifier(), $client);

    // Issue and persist access token
    $accessToken = $this->issueAccessToken($accessTokenTTL, $cli
ent, null, $finalizedScopes);

    // Inject access token into response type
    $responseType->setAccessToken($accessToken);

    return $responseType;
}
```

类似于授权码模式， `access_token` 的发放也是通过 `Bearer Token` 中存放 JWT。

密码模式

OAuth2 密码授权机制可以让你自己的客户端（如移动应用程序）邮箱地址或者用户名和密码获取访问令牌。如此一来你就可以安全地向自己的客户端发出访问令牌，而不需要遍历整个 OAuth2 授权代码重定向流程。

创建密码授权的客户端后，就可以通过向用户的电子邮件地址和密码向 /oauth/token 路由发出 POST 请求来获取访问令牌。而该路由已经由 Passport::routes 方法注册，因此不需要手动定义它。如果请求成功，会在服务端返回的 JSON 响应中收到一个 access_token 和 refresh_token：

```
$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'password',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'username' => 'taylor@laravel.com',
        'password' => 'my-password',
        'scope' => '',
    ],
]);

return json_decode((string) $response->getBody(), true);
```

只要用用户名与密码来验证合法性就可以发放 `access_token` 与 `refresh_token` ：

```
public function respondToAccessTokenRequest(
    ServerRequestInterface $request,
    ResponseTypeInterface $responseType,
    \DateInterval $accessTokenTTL
) {
    // Validate request
    $client = $this->validateClient($request);
    $scopes = $this->validateScopes($this->getRequestParameter('
scope', $request, $this->defaultScope));
    $user = $this->validateUser($request, $client);

    // Finalize the requested scopes
    $finalizedScopes = $this->scopeRepository->finalizeScopes($s
copes, $this->getIdentifier(), $client, $user->getIdentifier());

    // Issue and persist new tokens
    $accessToken = $this->issueAccessToken($accessTokenTTL, $cli
ent, $user->getIdentifier(), $finalizedScopes);
    $refreshToken = $this->issueRefreshToken($accessToken);

    // Inject tokens into response
    $responseType->setAccessToken($accessToken);
    $responseType->setRefreshToken($refreshToken);
```

```
    return $responseType;
}


protected function validateUser(ServerRequestInterface $request,
 ClientEntityInterface $client)
{
    $username = $this->getRequestParameter('username', $request)
;
    if (is_null($username)) {
        throw OAuthServerException::invalidRequest('username');
    }

    $password = $this->getRequestParameter('password', $request)
;
    if (is_null($password)) {
        throw OAuthServerException::invalidRequest('password');
    }

    $user = $this->userRepository->getUserEntityByUserCredential
s(
        $username,
        $password,
        $this->getIdentifier(),
        $client
    );
    if ($user instanceof UserEntityInterface === false) {
        $this->getEmitter()->emit(new RequestEvent(RequestEvent:
:USER_AUTHENTICATION_FAILED, $request));

        throw OAuthServerException::invalidCredentials();
    }

    return $user;
}
```

路由保护

Passport 包含一个 验证保护机制 可以验证请求中传入的访问令牌。配置 api 的看守器使用 passport 驱动程序后，只需要在需要有效访问令牌的任何路由上指定 auth:api 中间件：

```
Route::get('/user', function () {
    //
})->middleware('auth:api');
```

当调用 Passport 保护下的路由时，接入的 API 应用需要将访问令牌作为 Bearer 令牌放在请求头 Authorization 中。例如，使用 Guzzle HTTP 库时：

```
$response = $client->request('GET', '/api/user', [
    'headers' => [
        'Accept' => 'application/json',
        'Authorization' => 'Bearer '.$accessToken,
    ],
]);
```

auth:api 中间件

当我们已经配置完成 `Passport` 的四种模式并拿到 `access_token` 之后，我们就可以利用令牌去资源服务器获取数据了。资源服务器最常用的校验令牌的中间件就是 `auth:api` ，中间件是 `auth` ， `api` 是中间件的参数：

```
'auth' => \Illuminate\Auth\Middleware\Authenticate::class,
```

这个中间件是验证登录状态的常用中间件：

```
class Authenticate
{
    public function __construct(Auth $auth)
    {
        $this->auth = $auth;
    }

    public function handle($request, Closure $next, ...$guards)
    {
        $this->authenticate($guards);

        return $next($request);
    }


    protected function authenticate(array $guards)
    {
        if (empty($guards)) {
            return $this->auth->authenticate();
        }

        foreach ($guards as $guard) {
            if ($this->auth->guard($guard)->check()) {
                return $this->auth->shouldUse($guard);
            }
        }

        throw new AuthenticationException('Unauthenticated.', $g
uards);
    }
}
```

我们的参数 `api` 就是上面的 `guards`，`Auth` 是 `laravel` 自带的登录校验服务：

```
class AuthManager implements FactoryContract
{
    public function guard($name = null)
    {
        $name = $name ?: $this->getDefaultDriver();

        return $this->guards[$name] ?? $this->guards[$name] = $t
his->resolve($name);
    }

    protected function resolve($name)
    {
        $config = $this->getConfig($name);

        if (is_null($config)) {
            throw new InvalidArgumentException("Auth guard [{$na
me}] is not defined.");
        }

        if (isset($this->customCreators[$config['driver']])) {
            return $this->callCustomCreator($name, $config);
        }

        $driverMethod = 'create'.ucfirst($config['driver']).'Dri
ver';

        if (method_exists($this, $driverMethod)) {
            return $this->{$driverMethod}($name, $config);
        }

        throw new InvalidArgumentException("Auth guard driver [{
$name}] is not defined.");
    }

}
```

文档告诉我们，若想要使用 `passport` 服务，我们的 `config/auth` 文件需要如此配置：

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],

    'api' => [
        'driver' => 'passport',
        'provider' => 'users',
    ],
],
```

可以看出， `driver` 就是 `passport` ，我们在启动 `passport` 服务的时候曾经注册过一个 `Guard` ：

```
protected function registerGuard()
{
    Auth::extend('passport', function ($app, $name, array $confi
g) {
        return tap($this->makeGuard($config), function ($guard)
{
            $this->app->refresh('request', $guard, 'setRequest')
;
        });
    });
}

protected function makeGuard(array $config)
{
    return new RequestGuard(function ($request) use ($config) {
        return (new TokenGuard(
            $this->app->make(ResourceServer::class),
            Auth::createUserProvider($config['provider']),
            $this->app->make(TokenRepository::class),
            $this->app->make(ClientRepository::class),
            $this->app->make('encrypter')
        ))->user($request);
    }, $this->app['request']);
}
```

因此，`passport` 使用的就是这个 `TokenGuard` ：

```
class TokenGuard
{
    public function __construct(ResourceServer $server,
                                UserProvider $provider,
                                TokenRepository $tokens,
                                ClientRepository $clients,
                                Encrypter $encrypter)
    {
        $this->server = $server;
        $this->tokens = $tokens;
        $this->clients = $clients;
        $this->provider = $provider;
        $this->encrypter = $encrypter;
    }

    public function user(Request $request)
    {
        if ($request->bearerToken()) {
            return $this->authenticateViaBearerToken($request);
        } elseif ($request->cookie(Passport::cookie())) {
            return $this->authenticateViaCookie($request);
        }
    }
}
```

可以看到，`TokenGuard` 支持两种 `Token` 的验证：`BearerToken` 与 `cookie`。

我们首先看 `BearerToken`:

```
public function bearerToken()
{
    $header = $this->header('Authorization', '');

    if (Str::startsWith($header, 'Bearer ')) {
        return Str::substr($header, 7);
    }
}
```

```
protected function authenticateViaBearerToken($request)
{
    $psr = (new DiactorosFactory)->createRequest($request);

    try {
        $psr = $this->server->validateAuthenticatedRequest($psr)
;

        $user = $this->provider->retrieveById(
            $psr->getAttribute('oauth_user_id')
        );

        if (! $user) {
            return;
        }

        $token = $this->tokens->find(
            $psr->getAttribute('oauth_access_token_id')
        );

        $clientId = $psr->getAttribute('oauth_client_id');

        if ($this->clients->revoked($clientId)) {
            return;
        }

        return $token ? $user->withAccessToken($token) : null;
    } catch (OAuthServerException $e) {
        return Container::getInstance()->make(
            ExceptionHandler::class
        )->report($e);
    }
}
```

首先，需要验证请求的合法性：

```
class ResourceServer
{
    public function validateAuthenticatedRequest(ServerRequestInterface $request)
    {
        return $this->getAuthorizationValidator()->validateAuthorization($request);
    }

    protected function getAuthorizationValidator()
    {
        if ($this->authorizationValidator instanceof AuthorizationValidatorInterface === false) {
            $this->authorizationValidator = new BearerTokenValidator($this->accessTokenRepository);
        }

        $this->authorizationValidator->setPublicKey($this->publicKey);

        return $this->authorizationValidator;
    }

}
```

`BearerTokenValidator` 专门用于验证 `BearerToken` 的合法性：

```
class BearerTokenValidator implements AuthorizationValidatorInterface
{
    public function validateAuthorization(ServerRequestInterface $request)
    {
        if ($request->hasHeader('authorization') === false) {
            throw OAuthServerException::accessDenied('Missing "Authorization" header');
        }

        $header = $request->getHeader('authorization');
```

```
        $jwt = trim(preg_replace('/^(?:\s+)?Bearer\s/', '', $hea
der[0]));

        try {
            // Attempt to parse and validate the JWT
            $token = (new Parser())->parse($jwt);
            if ($token->verify(new Sha256(), $this->publicKey->g
etKeyPath()) === false) {
                throw OAuthServerException::accessDenied('Access
 token could not be verified');
            }

            // Ensure access token hasn't expired
            $data = new ValidationData();
            $data->setCurrentTime(time());

            if ($token->validate($data) === false) {
                throw OAuthServerException::accessDenied('Access
 token is invalid');
            }

            // Check if token has been revoked
            if ($this->accessTokenRepository->isAccessTokenRevok
ed($token->getClaim('jti'))) {
                throw OAuthServerException::accessDenied('Access
 token has been revoked');
            }

            // Return the request with additional attributes
            return $request
                ->withAttribute('oauth_access_token_id', $token-
>getClaim('jti'))
                ->withAttribute('oauth_client_id', $token->getCl
aim('aud'))
                ->withAttribute('oauth_user_id', $token->getClai
m('sub'))
                ->withAttribute('oauth_scopes', $token->getClaim(
'scopes'));
        } catch (\InvalidArgumentException $exception) {
            // JWT couldn't be parsed so return the request as is
```

```
            throw OAuthServerException::accessDenied($exception->getMessage());
        } catch (\RuntimeException $exception) {
            //JWR couldn't be parsed so return the request as is
            throw OAuthServerException::accessDenied('Error while decoding to JSON');
        }
    }
}
```

通过 `passport` 拿到的 `access_token` 都是 `JWT` 格式的，因此首先第一步需要将 `JWT` 解析：

```
class Parser
{
    public function parse($jwt)
    {
        $data = $this->splitJwt($jwt);
        $header = $this->parseHeader($data[0]);
        $claims = $this->parseClaims($data[1]);
        $signature = $this->parseSignature($header, $data[2]);

        foreach ($claims as $name => $value) {
            if (isset($header[$name])) {
                $header[$name] = $value;
            }
        }

        if ($signature === null) {
            unset($data[2]);
        }

        return new Token($header, $claims, $signature, $data);
    }

    protected function splitJwt($jwt)
    {
        if (!is_string($jwt)) {
```

```
            throw new InvalidArgumentException('The JWT string m
ust have two dots');
        }

        $data = explode('.', $jwt);

        if (count($data) != 3) {
            throw new InvalidArgumentException('The JWT string m
ust have two dots');
        }

        return $data;
    }

    protected function parseHeader($data)
    {
        $header = (array) $this->decoder->jsonDecode($this->deco
der->base64UrlDecode($data));

        if (isset($header['enc'])) {
            throw new InvalidArgumentException('Encryption is no
t supported yet');
        }

        return $header;
    }

    protected function parseClaims($data)
    {
        $claims = (array) $this->decoder->jsonDecode($this->deco
der->base64UrlDecode($data));

        foreach ($claims as $name => &$value) {
            $value = $this->claimFactory->create($name, $value);
        }

        return $claims;
    }

    protected function parseSignature(array $header, $data)
```

```
    {
        if ($data == '' || !isset($header['alg']) || $header['al
g'] == 'none') {
            return null;
        }

        $hash = $this->decoder->base64UrlDecode($data);

        return new Signature($hash);
    }

}
```

获得 `JWT` 的三个部分之后，就要验证签名部分是否合法：

```
class Token
{
    public function verify(Signer $signer, $key)
    {
        if ($this->signature === null) {
            throw new BadMethodCallException('This token is not
signed');
        }

        if ($this->headers['alg'] !== $signer->getAlgorithmId())
 {
            return false;
        }

        return $this->signature->verify($signer, $this->getPaylo
ad(), $key);
    }
}
```

验证通过之后，就要验证 `JWT` 各个部分是否合法：

```
$data = new ValidationData();
$data->setCurrentTime(time());
```

```
public function __construct($currentTime = null)
{
    $currentTime = $currentTime ?: time();

    $this->items = [
        'jti' => null,
        'iss' => null,
        'aud' => null,
        'sub' => null,
        'iat' => $currentTime,
        'nbf' => $currentTime,
        'exp' => $currentTime
    ];
}


public function validate(ValidationData $data)
{
    foreach ($this->getValidatableClaims() as $claim) {
        if (!$claim->validate($data)) {
            return false;
        }
    }

    return true;
}

public function __construct(array $callbacks = [])
{
    $this->callbacks = array_merge(
        [
            'iat' => [$this, 'createLesserOrEqualsTo'],
            'nbf' => [$this, 'createLesserOrEqualsTo'],
            'exp' => [$this, 'createGreaterOrEqualsTo'],
            'iss' => [$this, 'createEqualsTo'],
            'aud' => [$this, 'createEqualsTo'],
            'sub' => [$this, 'createEqualsTo'],
            'jti' => [$this, 'createEqualsTo']
        ],
        $callbacks
```

```
    );
}
```

我们前面说过，

- aud 客户端 id
- jti access_token 随机码
- iat 生成时间
- nbf 拒绝接受 jwt 时间
- exp access_token 失效时间
- sub 用户 id

因此，`JWT` 的生成时间、拒绝接受时间、失效时间就会被验证完成。

接下来，还会验证最重要的 `access_token` ：

```
if ($this->accessTokenRepository->isAccessTokenRevoked($token->getClaim('jti'))) {
    throw OAuthServerException::accessDenied('Access token has been revoked');
}

public function isAccessTokenRevoked($tokenId)
{
    return $this->tokenRepository->isAccessTokenRevoked($tokenId);
}

public function isAccessTokenRevoked($id)
{
    if ($token = $this->find($id)) {
        return $token->revoked;
    }

    return true;
}
```

接下来，`TokenGuard` 就会验证 `userid` 、`clientid` 与 `access_token` 的合法性：

```
$user = $this->provider->retrieveById(
    $psr->getAttribute('oauth_user_id')
);

if (! $user) {
    return;
}

$token = $this->tokens->find(
    $psr->getAttribute('oauth_access_token_id')
);

$clientId = $psr->getAttribute('oauth_client_id');

if ($this->clients->revoked($clientId)) {
    return;
}

return $token ? $user->withAccessToken($token) : null;
```

中间件验证完成。

客户端模式中间件 CheckClientCredentials

我们在上面可以看到 `auth:api` 中间件不仅验证 `access_token` ，还会验证 `user_id` ，对于客户端模式来说，由于 `JWT` 中并没有用户信息，因此 `passport` 专门存在中间件 `CheckClientCredentials` 来做非登录状态的校验。

```
class CheckClientCredentials
{
    public function handle($request, Closure $next, ...$scopes)
    {
        $psr = (new DiactorosFactory)->createRequest($request);

        try {
            $psr = $this->server->validateAuthenticatedRequest($
psr);
        } catch (OAuthServerException $e) {
            throw new AuthenticationException;
        }

        $this->validateScopes($psr, $scopes);

        return $next($request);
    }
}
```

使用 JavaScript 接入 API

在构建 API 时，如果能通过 JavaScript 应用接入自己的 API 将会给开发过程带来极大的便利。这种 API 开发方法允许你使用自己的应用程序的 API 和别人共享的 API。你的 Web 应用程序、移动应用程序、第三方应用程序以及可能在各种软件包管理器上发布的任何 SDK 都可能会使用相同的API。

通常，如果要从 JavaScript 应用程序中使用 API，则需要手动向应用程序发送访问令牌，并将其传递给应用程序。但是，Passport 有一个可以处理这个问题的中间件。将 CreateFreshApiToken 中间件添加到 web 中间件组就可以了：

```
'web' => [
    // Other middleware...
    \Laravel\Passport\Http\Middleware\CreateFreshApiToken::class
,
],
```

Passport 的这个中间件将会在你所有的对外请求中添加一个 laravel_token cookie。该 cookie 将包含一个加密后的 JWT ，Passport 将用来验证来自 JavaScript 应用程序的 API 请求。至此，你可以在不明确传递访问令牌的情况下向应用程序的 API 发出请求

```
axios.get('/user')
    .then(response => {
        console.log(response.data);
    });
```

当使用上面的授权方法时，Axios 会自动带上 X-CSRF-TOKEN 请求头传递。另外，默认的 Laravel JavaScript 脚手架会让 Axios 发送 X-Requested-With 请求头:

```
window.axios.defaults.headers.common = {
    'X-Requested-With': 'XMLHttpRequest',
};
```

CreateFreshApiToken 中间件

```
class CreateFreshApiToken
{
    public function handle($request, Closure $next, $guard = nul
l)
    {
        $this->guard = $guard;

        $response = $next($request);

        if ($this->shouldReceiveFreshToken($request, $response))
 {
            $response->withCookie($this->cookieFactory->make(
                $request->user($this->guard)->getKey(), $request
->session()->token()
            ));
        }

        return $response;
```

```
    }

    public function make($userId, $csrfToken)
    {
        $config = $this->config->get('session');

        $expiration = Carbon::now()->addMinutes($config['lifetim
e']);

        return new Cookie(
            Passport::cookie(),
            $this->createToken($userId, $csrfToken, $expiration)
,
            $expiration,
            $config['path'],
            $config['domain'],
            $config['secure'],
            true
        );
    }

    protected function createToken($userId, $csrfToken, Carbon $
expiration)
    {
        return JWT::encode([
            'sub' => $userId,
            'csrf' => $csrfToken,
            'expiry' => $expiration->getTimestamp(),
        ], $this->encrypter->getKey());
    }

    protected function shouldReceiveFreshToken($request, $respon
se)
    {
        return $this->requestShouldReceiveFreshToken($request) &
&
               $this->responseShouldReceiveFreshToken($response)
;
    }
```

```
    protected function requestShouldReceiveFreshToken($request)
    {
        return $request->isMethod('GET') && $request->user($this
->guard);
    }

    protected function responseShouldReceiveFreshToken($response)

    {
        return $response instanceof Response && ! $this->already
ContainsToken($response);
    }
}
```

这个中间件发出的 `JWT` 令牌仍然由 `auth:api` 来负责验证，我们前面说过， `TokenGuard` 负责两种令牌的验证，一种是 `BearerToken` , 另一种就是这个 `Cookie` :

```
public function user(Request $request)
{
    if ($request->bearerToken()) {
        return $this->authenticateViaBearerToken($request);
    } elseif ($request->cookie(Passport::cookie())) {
        return $this->authenticateViaCookie($request);
    }
}

protected function authenticateViaCookie($request)
{
    try {
        $token = $this->decodeJwtTokenCookie($request);
    } catch (Exception $e) {
        return;
    }

    if (! $this->validCsrf($token, $request) ||
        time() >= $token['expiry']) {
        return;
```

```
    }

    if ($user = $this->provider->retrieveById($token['sub'])) {
        return $user->withAccessToken(new TransientToken);
    }
}

protected function decodeJwtTokenCookie($request)
{
    return (array) JWT::decode(
        $this->encrypter->decrypt($request->cookie(Passport::coo
kie())),
        $this->encrypter->getKey(), ['HS256']
    );
}

protected function validCsrf($token, $request)
{
    return isset($token['csrf']) && hash_equals(
        $token['csrf'], (string) $request->header('X-CSRF-TOKEN'
)
    );
}
```